THE UNIVERSITY OF ALBERTA

The Design and Implementation of the Maple Virtual Machine

by

(C) Richard W. Gillespie

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES AND RESEARCH

IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE

OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTING SCIENCE

EDMONTON, ALBERTA

FALL, 1982

For my parents

Abstract

The Maple project involves the design and implementation of the various components of the Maple Programming System. This system is a complete programming environment through which programs written in the Maple language may interface with each other directly, rather than through a complicated operating system interface. The elements of the System are defined using the Maple programming language.

At the bottom level of the Maple Programming System is an abstract machine which is tailored to the Maple language and executes an intermediate form of Maple programs. The first part of the Maple project to be designed and implemented is an underlying machine. The major constructs of the Maple programming language are described by this thesis, and the requirements of a machine to execute expressions written in the language are outlined. The runtime garbage collection which is presented as part of the machine is of particular importance. A particular design for an abstract machine to run Maple programs is presented, together with the representation which the Maple language constructs have on the machine. Finally, the details of a pilot implementation of the Maple abstract machine, called the Maple Virtual Machine, are presented.

Acknowledgements

I would like to thank the members of my examining committee, Drs. Tony Marsland and Jeff Pelletier, for their time and efforts in shaping this thesis. I particularly want to thank my supervisor, Dr. Paul Voda, for his help and guidance during this work. As well, I wish to thank Chris Gray for his suggestions and criticisms.

Finally, my fond thanks to Anne whose help through the darker times was much needed, and whose affection and support was always there.

This work was supported, in part, by a grant from the National Science and Engineering Research Council of Canada.

Table of Contents

List of Figures

Chapter 1

Introduction

One of the fundamental areas of Computing Science is the study of high-level computer programming languages. The use of high-level languages reduces the time and effort involved in programming, eliminating concern for the minute details of machine language programming.

The design of programming languages has progressed a great deal due to the advances made in language theory, particularly during the 1960's [Tennent 1981]. Historically, programs have been executed within the environment of an operating system which is completely independent of the programming language. Recently, efforts have been made to integrate programming languages with their environment. This integration involves defining the surrounding system in terms of the features of the programming language, and allowing programs to have direct access to both the system and each other. The system then pre-defines some programs and data which are useful for user-defined programs. It is these pre-defined entities which supply a uniform environment for programs to run in. Also, this allows the programming language to be the command language for the programming system.

Since the early days of Computing Science, operating systems have been evolving from collections of disjoint pieces of software into unified programming systems which integrate elements of a programming language with the elements of an operating system. The programming language which such a system is based upon provides very general structuring mechanisms, but no specific structures. For example, such a language does not even provide pre-defined data types such as integers or characters. However, the programming system would provide definitions of those types for programmers to use. The distinction is that those types agree in definition and use with all other types which can be defined in the system.

In an integrated programming system there is no operating system *per se*. The user communicates directly with the compiler of the language, and expressions are compiled immediately upon definition. Programs in the system communicate directly,

with no complicated system calls necessary as an interface. Any program in the system is completely accessible to any other program in the system.

The programs of a traditional operating system map directly onto functions in an integrated programming system. A function is a body of code which may be executed many times, with different arguments each time. The action of executing the body of a function with a particular value bound to the function argument is called a *function application*. The process of running programs in a typical operating system corresponds to doing function applications in an integrated programming system. As well, the data files of an operating system map onto variables in an integrated programming system, since both are merely collections of data which can change.

The primary components of the Maple Programming System are the Editor, the Compiler, the Maple Tree, and the underlying machine. The Editor is the interface between the user and the rest of the Maple system, and is used to enter programs (functions) and data definitions into the Maple Programming System. These functions may be directly executed, or the user can have them stored in the Maple Tree so that they may be referenced later. The Editor uses the Compiler to have those functions and data definitions compiled and stored in the Maple Tree. The Editor and the Compiler have not yet been designed.

The Maple Tree is the equivalent of a file system in a traditional operating system. The Tree contains all of the functions and variables which are available to user-defined functions. These include the ones the provided by the system, plus the user defined ones. The grammar of the Maple language imposes a hierarchical structure on programs, and this is reflected in the structure of the Maple Tree. The Editor is the only means through which a user may modify the structure of the Tree by adding, changing, or deleting definitions in the Tree; a user program cannot change the Tree structure.

The execution of Maple expressions requires the machine which underlies the other components of the Maple Programming System. The machine has an architecture which is tailored to the structure of Maple programs so that Maple programs can execute quickly on it. An actual hardware implementation of the machine was not practical at this time, so a simulator, the Maple abstract machine, has

been designed and implemented to run Maple programs.

The design and implementation of a pilot version of the Maple abstract machine are presented by this thesis. An understanding of the Maple programming language is necessary before the requirements of the underlying machine may can be understood. A discussion of the major concepts of the language is presented first. The design of the Maple abstract Machine, and the the details of a particular implementation of the machine, the Maple Virtual Machine, are presented after the language description. Of primary interest in this discussion is the double stack model for performing runtime garbage collection. Finally, a summary of the results of the thesis is given, and some topics for future research are presented.

Chapter 2

The Maple Programming Language

The syntax and semantics of the Maple programming language are best described by first presenting the features of the Maple language, and the languages where those features originated, and describing Maple in terms of those features.

The description of Maple which is presented is not intended to be a comprehensive description of Maple, but is devoted mainly to the data structures of the language. The control structures of Maple are better described elsewhere [Voda 1982a]. The complete Maple language, including both syntax and semantics, is also given elsewhere [Voda 1982b].

2.1 Features of the Maple Language

2.1.1 Semantics

Most programming languages have their syntax designed first, and the semantics specified in an *ad hoc* fashion later. The result, usually, is a language whose semantics are very difficult to describe. When the semantics of a language are not well described, the writer of a compiler for the language has problems deducing the meanings of the language constructs.

LISP [McCarthy 1960] was the first programming language to be designed from a simple semantic basis (although there are some problems with bindings of names). Unfortunately, a pleasant 'syntactic sugaring' was not built on top of the semantic base. Consequently, the LISP syntax makes the language very difficult to use, primarily due to the proliferation of parentheses in the text of LISP programs.

Maple also is built from a simple semantic basis. However, the pragmatics of programming with the language were taken into consideration when the syntax was defined. Thus, Maple has a syntax which is easier to use than that of LISP, although the syntax of Maple is still quite different from that of more traditional block-structured languages such as Pascal [Jensen and Wirth 1975].

2.1.2 Types and Generic Functions

The notion of the definition of data types was first introduced by Algol68 [Pagan 1976] and popularized by Pascal. The abstract data types of Pascal allow a programmer to give a name to some data structure. That name is then equivalent to the data structure to which it is bound. For example, if a stack were to be represented by an array and a index into the array for the top of the stack, then the name 'stack' can be bound to that representation. After the definition of the stack representation, the name 'stack' can be used in place of the actual representation.

When Pascal was first described, its type definitions were adequate. However, the restrictions imposed by Pascal (functions cannot return large structures, functions cannot be part of structures) are now considered to be too limiting. The language Russell [Boehm, Demers, and Donahue 1980; Demers and Donahue 1980] expanded on Pascal types by generalizing function definitions and allowing types to be considered values which may be manipulated in the language. Lawine [Swierstra 1980] also has generalized types.

As a consequence of generalizing the types which functions can return, and the types of function parameters, generic functions may be created. A *generic* function is a function which can be parameterized by a type, rather than only by values. Generics are useful for reducing the redundant programming made necessary by languages without generics. For example, to implement stacks of integers in a language without generics one would design an abstract data type to represent stacks of integers and then design functions to manipulate that type. However, if stacks of characters were needed also, then it would be necessary to duplicate the definition of the integer stacks and their functions, replacing all uses of 'integer' by 'character'. This would involve a great deal of code and would also lead to the problem of giving names to the functions for the different stacks. On the other hand, a generic function for stack manipulation would have the type of the elements in the stack as a parameter, and such a function would be able to manipulate any type of stack.

Because of Pascal's limitations on function parameters and return types, generic functions are not possible in Pascal. Since Russell allows types to be treated as values, generic functions were included in the language easily. Lawine also has

generic functions as part of the language. The language Ada [Barnes 1980] also claims to have generic functions; however, the generics of Ada are actually just compile time macros.

Maple has generalized types and generics similar to those of Russell. Russell suffers from the syntactic problem that the programmer must specify the types of variables and functions far more often than is desirable. In Maple the types of values can be inferred, or derived, from the context of use. Thus, Maple does not suffer from the 'over-typing' of Russell.

2.1.3 Classes

The concept of classes was first introduced in the language Simula [Dahl *et al.* 1973]. A *class*, also known as a *definition procedure* [Tennent 1981], is a procedure whose body provides a set of definitions. The body of a class generally consists of a set of variable and function definitions. The variables within the class may only be accessed by use of the functions defined in the class. Execution of a class produces an *instance* of the class. An instance consists of locations for the variables of the class and copies of the functions. For example, a class to implement a stack would contain some internal representation of the stack, say an array plus an index into the array to represent the top of the stack, and functions to manipulate the array representation, say 'top', 'pop', and 'push'. An instance of this stack class would consist of space allocated to store the array and index, and a copy of each of the functions.

When an instance is created it is bound to some name. The functions associated with the instance can then be *selected* from that name in a fashion similar to record selection in Pascal. If 'c' were an instance of the stack class, then the selection 'c.top' would call the function 'top' associated with instance 'c'.

Smalltalk Classes

The Smalltalk programming language [Goldberg 1981] [1] is the one language which Maple most closely resembles. One of the features which Smalltalk and Maple share is the class concept.

---

[1] The August 1981 issue of Byte magazine was devoted to the Smalltalk programming language. A more easily found, but out-of-date, description of Smalltalk may be found in [Ingalls 1978].

The Smalltalk designers [Xerox Learning Research Group 1981] call Smalltalk an object-based language. An *object* in Smalltalk is a collection of data and/or functions. Thus everything in Smalltalk is an object. A *variable* in Smalltalk is an object which represents some storage location and whose contents are allowed to change.

A class in Smalltalk is a data structuring mechanism rather than a definition procedure. A Smalltalk class is an object which provides a template for the structure of other objects, called instances. This template consists of the definition of variables and *methods* (method is the Smalltalk word for function). There are some variables in a class definition which are local to each instance, called *instance variables,* and other variables which are local to methods, called *method variables,* which exist only while the method is being executed. Every instance has its own copy of instance variables which make up the internal representation of the instance. A class also contains a method which belongs only to the class and not to the instances. This method, called 'new', is used to create new instances. All of the other methods defined by a class belong to the instances of the class. For example, the definition of a 'stack' class in Smalltalk would consist of the name 'stack', the instance variables 'a' and 't' (for array, and top index), and the methods 'top', 'pop', and 'push'. All of these are available to any instance of the class 'stack'. As well, 'stack' defines a method 'new' which creates new stack instances by allocating memory locations for 'a' and 't', and associating the three stack functions with those memory locations.

The process of calling a method of an object is referred to as "sending a message". The message consists of the name of the object, the name of the method, and arguments (if any are required). The value returned by a method is another instance, although not necessarily an instance of the same class.

A Smalltalk class may be a subclass of another *superclass.* A subclass is obtained by modifying the definition of some part, or parts, of the superclass. Note that the superclass of one class may, again, have another superclass. Ultimately all classes are subclasses of a particular superclass, called **Object**, which has no superclass of its own.

Although the definition of classes is hierarchical, there is no hierarchy among the instances of a class. In fact, the Smalltalk language does not impose any

organization on any objects which are created. In a running Smalltalk system there is no need for such organization since the user needs only to mention the name of an object and the system finds it. This does imply, however, that the Smalltalk system must impose a hidden organization on the objects.

Maple is not an object-based language like Smalltalk, but does have a similar class data structure. However, the classes of Maple are not based on a simple class hierarchy, the superclasses, as those of Smalltalk are. A class hierarchy is possible in Maple, but is of limited use. As well, Smalltalk classes have *class variables* which are available to every instance of a class but only one copy of those variables is ever allocated. Maple has no such variables. Although Maple doesn't have the class hierarchy it does impose a hierarchy on the structure of programs.

2.1.4 Integrated Languages and Environments

Another feature which Maple shares with Smalltalk is the integration of the programming language with its environment to produce a programming system. In effect, a programming system, which is tailored to a particular language, fulfills the role which is usually filled by an operating system. The result is that the command language for the programming system is the same as the programming language, possibly with a few additions. Thus, programs may interface directly with the environment in which they are running.

Both the Smalltalk and Maple systems supply definitions of the functions and variables which provide the environment for programs. Whenever something needs to be done at the system level (eg. input or output), there is either a function already defined to do it or such a function may be built from the predefined functions.

2.2 Maple

The Maple programming language introduces a hierarchical structure to Maple programs which is not unlike that of the directory system of the UNIX operating system [Ritchie and Thompson 1974]. This entire collection of Maple 'programs' is referred to as the Maple Tree. The Maple Tree contains all predefined functions and variables which are pre-defined by the Maple system, as well as the user

defined functions and variables. The predefined section of the Maple Tree includes such things as arithmetic functions, array constructors, strings, text formatters, and the input/output functions. Each node of the Maple Tree is a Maple expression which has a name associated with it. Every node in the Tree has a special name associated with it, called **up**, which refers to the 'father' node of that node. The organization of the Maple Tree is best explained in terms of the concepts introduced by the programming language.

2.2.1 Introduction to the Maple Programming Language

Before describing the Maple language in detail some of the general concepts of the language must be introduced.

There are actually two Maple programming languages. There is a strongly typed and restrictive language, known as the *strict* Maple language, which is easy to describe. However, that which is easy to describe is not always easy to use. There is also an *extended* Maple language which is easier to use than the strict language. The extended language is defined from the strict language by the introduction of abbreviations into the syntax and semantics. The extended language relaxes the specification of type information by having the compiler infer types from context. The result is that the extended Maple language has a highly context-sensitive grammar. This discussion deals with the extended language because it is easier to use and read. Appendix A gives the syntax for the strict language. The abbreviations which lead to the extended language are given elsewhere [Voda 1982b].

All statements which can be formed from the grammar for the Maple language are expressions. Maple is a strongly typed language, the strict Maple language especially so. An expression in Maple may be either a type or a value. A type expression must be used in the so-called type positions of the Maple syntax. If an expression is a value then it also has a type associated with it, and a value expression must explicitly state what its type is. The extended Maple language relaxes the type specification by having the compiler derive the type of an expression from the context of the expression. Expressions which are types are for use by the compiler for performing type checking, and cannot be manipulated by the

machine at runtime.

The Maple language has two chief data structuring constructs: *groups* and *classes*. Groups are used to impose a hierarchical structure on the Maple Tree, while classes provide the general format for the definition of storage structures. An internal node of the Maple Tree is either a group or a class. The leaf nodes of the Maple Tree are either *elements* (instances) of classes, or functions.

The strict Maple language includes six different kinds of clauses. One of these, the **use** clause, introduces temporary values into Maple expressions. The rest of the Maple clauses are for control of flow. These are: **case** clauses, **selection** clauses, **par**allel execution clauses, and two exception handling clauses, **fail** and **attempt**. The last three (**par**, **fail**, and **attempt**) are not described in this thesis, since this discussion deals primarily with the data structures of Maple, and these last three clauses are not important to the definition and manipulation of data structures in Maple. The extended Maple language introduces further control of flow clauses which are defined in terms of **case** clauses and the predefined boolean class. These additional boolean clauses include the **if**, **while**, and **for** statements.

2.2.2 Maple Groups

In Maple, groups are expressions which are used to introduce structure into the Maple Tree. A *group* may be considered a 'directory' in the Maple Tree hierarchy, and is used to collect several disjoint field values (or types), to constuct a value group (or type group).

The syntactic definition of a group is similar to a record declaration in Pascal. In Maple syntax, a group definition is a list of fields, separated by semi-colons and enclosed within square brackets. Each field has a name, called the selector of the field, followed by a type, and possibly by a value on the right side of a colon. In the extended language the type need not be specified if it can be derived by the compiler. Note that a Pascal record consists strictly of fields which are storage values. However, the fields of a Maple group may be classes, elements, functions or groups. Any field which is not a function field is called a proper field.

The following example defines a group, 'g', which contains three fields:

```
g : [ a is int : 10 ;
      b with char is int : . . . ;
      c is [ . . . ] : [ . . . ] ;
    ]
```

The name of this group is 'g', and it contains the three field selectors 'a', 'b', and 'c'. Field 'a' is declared to be an integer and is initialized to the value 10. Field 'b' is a function which takes a character as an argument and returns an integer as the value of the function. Finally, 'c' is another group. The details of the values of 'b' and 'c' have not been given. The value of field 'a' of this group is obtained by the selection 'g.a'.

As mentioned before, a group represents either a value group or a type group. A value group is a group in which every field has a type and a value. A type group is a group in which every field has only its selector and type specified. For example, the following is a type group:

```
m is [ x is int ;
       y is real ;
       z with int is int ;
     ]
```

There is a special group, called the *empty* group, which has no fields. This empty group, [ ], can be used as either a type group or a value group.

The declaration of a group causes the Maple Editor to create new nodes in the Maple Tree. These nodes correspond to the fields of the group, and are labelled with the selector names of the fields.

2.2.3 Maple Classes

The definition of a value group, 'g', allocates the storage for the value of the group immediately. The value of the group is the collection of values of the fields of 'g'. However, if one wants another group which has the exact same structure as 'g', it is necessary to duplicate the definition of 'g', or to define a function which returns a group as its value. Since the repetition of group definitions is tedious and the definition of group-returning functions is cumbersome, the Maple language has a mechanism for defining 'templates' of storage structures without having the storage allocated immediately; this mechanism is provided within the Maple class definition.

A Maple class contains one or more **sort** fields. A **sort** field of a class definition contains a *base* type, which is hidden by the class, and an *apparent* type, which is the how the elements of the class appear outside of the class definition. An element of a class is stored as a value from the base type of the class, but the class hides this base type from use outside of the class. The base type value of an element of a class may only be manipulated through the use of the functions which are defined within the class. When an element of a class is created, there is storage allocated to store a single value from the base type of the class. The functions defined by the within the **sort** field of a class are associated with each element which is allocated, so that those functions may be selected from elements of that class. Note that an element may be a *variable* element whose underlying storage value is allowed to change, or a *constant* element whose storage value is not allowed to change. Any number of elements of a class may be allocated.

The definition of a class looks just like a group definition except for the **sort** fields. The declaration of a class consists of two parts: 1) the **sort** fields, and 2) some functions. A **sort** field consists of a field selector, the keyword **sort**, a **sort** group, and the base type of the **sort** field. The **sort** group contains the definition of functions which are associated with all elements of that class. These functions are called *element* functions because they may refer to the value of the element from which they are selected by the name **elem**. The following example gives the definition of a class representing the powers of 2 by their logarithms. Note that there are easier ways to represent the powers of 2, but this example serves to illustrate the principles of classes:

```
power : [ t : sort [ next with [ ] is power.t : elem + 1 ;
                     log with [ ] is int : elem ;
                     toInt with [ ] is int : 2 ** elem ;
                   ] as int ;
          first is power.t : 0 ;
          new with [ ] is var power.t : int.new [ ]
        ] ;
```

The class 'power' defines and hides the base type 'int' (**as** int). The class contains a particular value from the base type ('first') as a starting value for elements of the class, and a function to create new elements of the class ('new'). Notice how the allocation of a new element of class 'power' is done by allocating a new value

from the base type 'int'. In addition to a value of the base type, every element of the class 'power' has three functions associated with it: 1) a function which returns the next 'power' after that element by adding one to its base type value ('next'), 2) a function to convert the base type value of an element to the external type 'int' by simply returning the value of that element ('log'), 3) a function to raise 2 to the power of the base type value of an element, and return the 'int' type value ('toInt'). Note that only the functions 'next', 'log', and 'toInt' are associated with an element of class 'power'.

The definition of the class 'power' builds the functions once, but doesn't allocate any memory for any values of the base type of the class. An element of the class 'power' contains storage for a value from the implicit set, and has the element functions defined in the **sort** group associated with it. No elements are built until the function 'new' is invoked.

Each **sort** field of a class provides the definition of a new apparent type in terms of a base type. The apparent type of a **sort** field is the type of the **sort** group. For example, the apparent type of the **sort** field of class 'power' is:

```
[ next with [ ] is power.t ;
  log with [ ] is int ;
  toInt with [ ] is int ;
]
```

The base type of a **sort** field must be from another class. The function 'new' of a class then makes use of the 'new' function of the class for the implicit set. For example, the 'new' of the class 'power' is defined in terms of the 'new' of the 'int's (which are pre-defined in the Maple Tree). The keyword **sort** was chosen because a **sort** field defines a new type in the language, and not because there is some hidden ordering. **sort** is simply a synonym for **type**.

The functions fields of the **sort** group of a class are called element functions because when they are selected from an element they may refer to the value, from the base type defined by that **sort** field, of that element. Since element functions can refer to the element they are selected from, binary and post-fix functions may be built. Element functions are also discussed in section 2.2.4. The functions of a class which are not defined within a **sort** group are used to define new elements of the class and to manipulate existing elements of the class.

Note that the Maple programming language allows a class to contain more than one **sort** field. However, having more than one **sort** field associated with a class is of unknown practical value.

A class value is a class in which each field of the class has a value. A **sort** value is the value of the **sort** group followed by the keyword **as** and the type of the implicit set of that **sort** field. A class type is a class in which only the types of the fields are specified. A **sort** type is the type of the **sort** group of the **sort** field.

2.2.4 Maple Functions

Functions in Maple are always fields within a group or class. When a function is declared, an expression or expression list makes up the body of the function. The value returned by a function is the value of the last expression of the function body. (It is important to distinguish between the return value of a function and the function value: the function value is the body of the function.) A function field consists of: the keyword **with**, the type of argument which the function accepts, the keyword **is**, the type of the value returned by the function, and the function body. Note that both the argument and the return value can be of any Maple type, except that a function cannot return another function as its result. It can, however, return a group which contains function fields. For an example of a function definition see function 'next' in the example class of section 2.2.3. The types of both the argument and the return value of 'next' are 'power.t'. The value returned by 'next' is the value of its argument plus one.

The use of a function in a program is called a function application. A function application consists of the function name followed by the argument to the function. The semantics of Maple insist that the argument be of the proper type.

A Maple function always has one argument. If more than one argument is required then they must be combined into a single group argument. If no arguments are desired then the empty group is used as both the argument type in the definition and the value of the argument when the function application is done. The argument is always referred to by the name **arg** within the function body.

There are special kinds of Maple functions called element functions. Such an element function may refer to the element from which the function has been selected. Within the body of an element function the value of the element from which the function was selected is referred to by the name **elem**. An element function can be declared only within a **sort** group of a class. The type of **elem**, in an element function, is always the base type of that **sort** field. A function which is not an element function is called a *single argument* function. See the 'power' class of section 2.2.3 for examples of element functions.

The definition of a function creates some new names which may be used within the body. In the case of an element function the name **elem** is also added to the new environment. **elem** refers to the element from which the function was selected. The function application 'f a', where 'f' is a function and 'a' is an expression, is executed by first evaluating the expression 'a'. The body of function 'f' is then executed with the value of expression 'a' as the argument. Note that the name **up**, when used within the body of a function, refers to the part of the Maple Tree where the function is defined and not where the function is used.

The argument to a Maple function is not limited to values but can also be a type. A function which takes a type as an argument and returns a group containing functions to manipulate values of the specified type is a generic function.

2.2.5 Maple Records

The Maple Tree provides the pre-definition of many useful classes such as 'int', 'char' and 'real' which contain their own 'new' functions to allocate storage. Thus, any class, whose base type is one of these pre-defined classes, can use the 'new' functions of that class to allocate new elements. However, often a user will want the values of the base type of a class to be composite values, or n-tuples of values. For example, the implicit set of values of a stack will have an array and an index for the top of the stack. However, the user cannot declare such a class without the aid of the language, since it is not possible to then specify the body of the 'new' function for the class.

The Maple language provides a record construct, called **rec**, which is a short-form way of defining classes. **rec**, when given a type group as an

'argument', is equivalent to a class whose implicit set of values are value groups. **rec** also supplies the definition of a function 'new' to create new elements of that record. **rec** also defines some other useful functions which are used to assign values to elements of that record, and to compare two elements of that record.

Consider the following **rec** definition and the type of the class which the record is equivalent to:

```
r : rec [ x is int ; y is char ]

    is equivalent to the class type

r is [ type is sort [ := with r.type is [ ] alter ;
                      ? with r.type is order ;
                      x is var int;
                      y is var char;
                    ] ;
       new with [ ] is var r.type;
       coerce with [ x is int ; y is char ] is r.type ;
     ] ;
```

Note that the bodies of these functions cannot be specified in the Maple programming language.

The keyword **alter** indicates that that function field is only available to those elements which are variables, i.e. those elements created with the 'new' function of the class. When **var** is specified on a proper field it indicates that that field is a variable of the specified type if and only if the entire element is a variable; otherwise the field is a constant of the type. The base type of the record, which is not specified here, is chosen by the compiler but, at least, includes the two fields 'x' and 'y'.

The class which is created by **rec** includes some special functions which cannot otherwise be declared in the Maple language. These functions are:

a. assignment (':=') to a variable element,

b. comparison ('?') of two elements of the same class,

c. creation ('new') of a new variable element of the class,

d. creation ('coerce') of a new constant element of the class.

The two functions := and ? are element functions, i.e. they are defined within the sort field of the class returned by the **rec** construct and manipulate the value of the element from which they are selected. The comparison function compares two elements and returns a 'value' of equals ('='), less than ('<'), or greater than ('>'); these three values are represented by an *enumerated type* as described below.

Since the elements of Maple records can be made up of composite values (in the implementation type), the comparison of two elements is done in a lexicographic fashion. The order of the components in the group of the record definition is the order defined for the lexicographic ordering on the elements, i.e. the first component has priority of the second component etc. For example, if two elements, 'a' and 'b', of the record 'r' above had the implicit values [3 ; "m"] and [3 ; "q"], respectively, then element 'a' would be less than element 'b' (since 3=3 and "m" < "q"). In general this ordering is not meaningful for the record elements, but there is no known ordering which actually is meaningful for all elements. The advantage of using a lexicographic ordering is that it is well-defined.

Record definitions are particularly useful as the types of the implicit set of a class, when it is necessary for the values to be n-tuples. For example, a class to represent stacks of integers would look like:

```
stack : [ t : sort [ top with [ ] is int : . . . ;
                     . . . . ;
                   ]
          as rec [ a : array [ 1 ; 10 ; int ] ;
                   tp : int
                 ] ;
        new with [ ] is var stack.t : . . . . ;
        . . . . ;
      ] ;
```

The use of a record as the implicit set of values for this class allows the implicit value of an element to have two components: an array, 'a', and an index for the top of the array, 'tp'. An element of 'stack' could contain at most 10 integers in its internal array. The body of function 'new' would use the 'new' function created by the **rec** to contruct new elements of class 'stack'. The user could also define functions to assign to a whole 'stack' element at a time, or to compare two 'stack' elements with the use of the functions ':=' and '?' defined by the **rec**.

Occasionally, it is desirable to have structures which are equivalent except for a few fields. Rather than defining many different records to represent these structures, it is possible to combine them into a single record definition containing *variants*. The variants of a record define a kind of set of logicals, or booleans, one logical for each variant of the record. Although the record definition contains all of the variants, an element of that class will have only one variant which is 'on' and the rest of the variants will be 'off'. The variant which is actually 'on' for an

element is called the *state* of that element. Note that the state of an element (the variant which is 'on') may be changed as the result of a function application involving that element.

A record which contains variants may or may not have *fixed* fields which are the fields that all elements of that record will contain at all times. Syntactically, the fixed fields all come before the variants. All variants have a variant tag selector and, possibly, variant fields. A variant may have any number of variant fields, and those variant fields must either be proper fields or function fields. A variant tag is the symbol "!" followed by any legal Maple symbol (see [Voda 1982b]). For example, any married person has a name and an age, but may be either male or female. A male has a wife who has a name, and a female has a husband who has a name. Rather than defining two different records for representing males and females, a record with variants may be used:

```
vr : rec [ Name is string ;
           Age is int ;
           !male WifeName is string ;
           !female HusbandName is string ;
         ] ;
```

This record is equivalent to the Pascal type:

```
type vr = record
             Name : string ; Age : integer;
             case t : ( male, female ) of
                male : ( WifeName : string ) ;
                female : ( WifeHusband : string ) ;
          end ;
```

The Maple class type constructed by the above record is:

```
vr is [ type is sort [ := with vr.type is [ ] alter ;
                       ? with vr.type is ordered ;
                       Name is var string ;
                       Age is var int ;
                       !male WifeName is string ;
                       !female HusbandName is string
                     ] ;
        male with [ Name is string ;
                    Age is int;
                    WifeName is string
                  ]
             is !male vr.type ;
        female with [ Name is string ;
                      Age is int;
                      HusbandName is string
                    ]
               is !female vr.type ;
        new with [ ] is var vr.type ;
      ] ;
```

The two functions 'male' and 'female' (outside of the **sort**) create and initialize

constant elements in the states 'Imale' and 'Ifemale' respectively.

The record class constructed by **rec** contains a function for each variant tag. These functions create new elements which are initially in the state of that variant tag, although they may be changed as a side-effect of some other function. A record containing variants does not have a function 'coerce'.

The variant fields of an element may only be selected from an element if it is known at compile time that the element is in the proper state. It is this protection which separates Maple from Pascal. Pascal does not enforce any type checking on variants, the Pascal designers merely state that to select variant fields at the wrong time is potentially dangerous. The type of a variant of a class is the type of the variant field(s) preceded by the variant tag.

The **case** clause of Maple allows the user to distinguish the state of an element. When the name of the element is used in a **case** clause the state of the element is returned as a variant tag. The returned state is then used as the discriminator for a series of expression lists, each of which is preceded by a variant tag. If there is no list preceded by the appropriate variant tag then a special list, headed by the keyword **out**, is executed. If the variable 'person' were an element of the record 'vr' defined above, then the following **case** clause must always be done in order to legally manipulate any variant fields of 'person':

```
case person in
   Imale . . . . ;
   Ifemale . . . . ;
end ;
```

A Maple record which contains only variant tags, and no fixed or variant fields, is called an *enumerated type*. Enumerated types fulfill the same role as enumerated types in Pascal. Since there are no fields associated with the variants, the names of the variants (without the "I" symbol) are available as values of the enumerated type, rather than as functions. The class 'ordered', which is predeclared in the Maple Tree, is such an enumerated type. This class consists of the three variant tags 'I<', 'I=', and 'I>'. The comparison function created by the **rec** construct uses this 'ordered' class for the result of comparisons. The boolean class defined in the extended Maple language is also based upon the 'ordered' class.

A **case** clause may also use the value of an expression as the discriminator, if that value is a variant tag. For instance, using the comparison function '?' of the integers and the class 'ordered':

```
case xyz.? 0 in
   !< xyz.:= 0;
   != xxx.:= 0;
   !> yyy.:= -5;
end ;
```

If the value of 'xyz' is less than 0 then the value 0 is assigned to 'xyz'. If the value of 'xyz' equals 0 then 'xxx' is assigned 0. Finally, if 'xyz' is greater than 0 then 'yyy' is assigned the value −5. (Note that in the extended Maple language the selection operator "." may be omitted from selections; the compiler automatically inserts the operator.)

case values and **case** types are more complicated to explain than the values and types encountered so far. **case** values and types are defined in terms of the union values and types of the expression lists making up the **case** clause. These unions are beyond the scope of this discussion and are best described elsewhere [Voda 1982b].

2.2.6 Maple Use Clauses

Maple **use** clauses correspond to the blocks of Algol, and are used to introduce new, temporary values into Maple expressions. The form of a **use** clause is:

use LocalValue **in** Body.

This corresponds to the pidgin Algol block:

```
begin
   declare loc ;
   loc := LocalValue ;
   . . . Body . . . ;
end ;
```

Both 'LocalValue' and 'Body' are Maple expressions. The value of the expression 'LocalValue' is bound to the name **loc** within the body of the **use** clause. As with functions, the result of a **use** clause is the value of the last expression within the body of the clause. The value of **loc** cannot be referenced from outside of the **use** clause, although **loc** may be used to form the result of the **use** clause.

It is interesting to note the close relation between **use** clauses and functions. The **use** clause "**use** L **in** B", may also be represented by the function application:

[(x **with** Ltype **is** Btype : B')].x L

The function body B' is the same as B except all occurrences of the name **loc** are replaced by the name **arg**. Ltype and Btype are the types of expressions L and B respectively.

The **use** clauses of Maple are particularly useful for the definition of packages and generic functions as described in [Voda 1982a]. A package is a **use** clause which returns a group which contains the local value(s) declared by the **use**. This is due to the fact that the value local to the body of the **use** may remain bound to the result of the **use**. For example,

use int.new[ ] **in** loc := 0; loc **end**

is a **use** clause which declares an integer variable, assigns zero to it, and then uses that variable as the value of the **use**. This example demonstrates a **use** which leaves its local value bound after the clause is finished. Other applications of **use** clauses to construct packages are given in [Voda 1982a]. The fact that the local value can be bound to the return value of a **use** clause has repercussions in the garbage collection of a running system. Not all local values can be thrown away after the **use** clause has been evaluated, as is done with Algol blocks. In fact, the same holds true for functions in Maple, since the argument to the function may remain bound to the value returned by the function. The issues of local values which remain bound, and garbage collection are addressed in Chapter 3.

2.2.7 Maple Names

A name in the strict Maple language must be fully described by the *name path* which leads to that node in the Maple Tree. A name path is a series of selections of field selectors from groups, and follows a 'path' through the Maple Tree. The selections start either at the current environment, indicated by the special name **env**, or at the root of the Maple Tree, indicated by the special name **root**. The names in the selections must either be **up** (to move up the Tree), or the field selector names of a group (to move down the Tree). In general, a user will not want to bother to fully qualify names in this manner because it becomes quite

tedious. The extended Maple language allows the user to abbreviate names by not specifying the complete path. For such abbreviated names the Maple compiler is responsible for 'finding' the appropriate node in the Maple Tree. The rules for finding such names are given elsewhere [Voda 1982b].

2.2.8 The Maple Tree

The Maple Tree provides the integrated environment for Maple programs. The Maple Tree is a group which includes all the predefined functions and variables. From the very top, the Maple Tree might look something like this:

```
root : [ standard :
             [ order : rec [ !< ; != ; !> ] ;
               precision : rec [ !single ; !double ; !many ] ;
               fixed with precision : . . . . ;
               float with precision : . . . . ;
               int : fixed single ;
               real : float single ;
               char : . . . . ;
               array with [ . . . ] : . . . . ;
               seq with sort [ . . . ] : . . . . ;
               string : seq char ;
               . . . . ;
             ] ;

         nonstandard :
             [ . . . . ] ;

         prog :
             [ . . . . ] ;

         user :
             [ paul : [ . . . . ] ;
               jeff : [ . . . . ] ;
               tony : [ . . . . ] ;
               . . . . ;
             ] ;

         . . . . ;
       ] ;
```

The group 'standard' includes the predefined 'types' which are useful for programmers. Note that for efficiency's sake these are generally programmed in the machine code of the host computer.

The group 'nonstandard' contains the installation dependent programs and variables for a particular Maple system. The group 'prog' contains the most heavily used functions in the Maple Tree, and corresponds to the '/bin' directory of UNIX. Finally, the group 'user' contains all the users of the Maple system and their individual functions and variables.

Every node in the Maple Tree has a name and is either a group, class, function, or element. Groups and classes form the internal nodes of the Tree, while functions and elements are the leaf nodes of the Tree. Any node in the Maple Tree may refer to any other node in the Tree by specifying the name path of the other node.

2.3 Summary

This chapter has briefly described the Maple programming language. This language shares such features as simple semantics, generalized types, classes, and integrated environment with other languages which have been designed in the past. The Smalltalk language, in particular, is quite similar to the Maple language.

The description of the Maple language presented in this chapter has emphasized the data structures of Maple rather than the control of flow clauses. The representation of Maple data structures turns out to be of prime importance in the design of the Maple abstract machine.

Chapter 3

The Design of the Maple Machine

A compiler for a programming language must generate machine code to be run on a target computer architecture. If the code for the target machine can be generated easily, then the compiler may be designed and implemented quickly. In order for the code generation to be done simply, the compiler's internal representation of the code and the external representation in machine code must correspond closely. There are two approaches used to achieve this similarity: the internal representation is forced to match the machine representation, or vice versa. Unfortunately, the first solution results in a reduction in the portability of the compiler since the compiler is designed for a particular architecture. The second solution, on the other hand, requires that a new machine be designed and built to run the code generated. But, rather than building such a machine with hardware, it can be simulated in software on any given host computer. A simulator for such an abstract machine accepts instructions for the target machine as input and executes those instructions on the particular host computer. A simulator which is tailored for a particular high level language is called an *abstract machine*.

It is especially attractive to use a simulator when the programming language design is still evolving. It is far easier to change a small part of a program which simulates a machine than to change a hardware implementation in order to facilitate some new feature of the language. A program is also easy to experiment with, since changes to the simulated architecture can be made quickly and the result noted. Unfortunately, the price paid for these advantages is slow execution time. Once the language definition has stabilized and the best machine architecture realized, the abstract machine can be implemented in firmware, as microcode, or even hardware, as an actual machine, to speed up execution.

Since the Maple programming language has not stabilized, and it is not clear what architecture is best for running Maple programs, a preliminary abstract machine has been designed. The design of this abstract machine was influenced by

the machine designs for Smalltalk and Lawine.

3.1 The Smalltalk Machine

A complete Smalltalk system consists of several distinct pieces, such as an editor, a compiler, and a debugger. Most of these may be written in Smalltalk itself because there is an abstract machine underlying the Smalltalk system. This Smalltalk Machine [Krasner 1981] consists of two primary parts: a storage manager, and an interpreter of the Smalltalk instructions.

The storage manager for the Smalltalk Machine is tailored to the Smalltalk language by being based upon Smalltalk objects. Memory references within the machine are in terms of object descriptions; these object descriptions may be of any size. This allows the storage manager to optimize space allocation, but requires extra bookkeeping to keep track of where each object is located. As well, in a running system, memory tends to become fragmented as objects are created and destroyed.

The Smalltalk interpreter is a stack-oriented machine which executes the *bytecodes*, or instructions, emitted by the Smalltalk compiler. These bytecodes instruct the interpreter to:

a. push an object description onto the stack,
b. store the value on the top of the stack in a variable,
c. pop the top of the stack,
d. branch (conditionally or unconditionally),
e. send a message to an object,
f. return the value on the top of the stack as the result of the current method.

Note that performing any of these operations may take one or more bytecodes.

The Smalltalk interpreter stack is used for standard stack operations, such as pushing, popping, and storing results. It is also used for calling functions and passing parameters. Sending a message to a method of an object, which corresponds to calling a routine on typical machines, is done by specifying the selector (name of the method) and the object which that selector belongs to, rather than by giving the address of the code to be executed. The interpreter

maintains a dictionary which associates the names of methods with the locations of those methods in memory. The stack is used to temporarily store the object and the arguments of a message, and to hold the resulting return value.

Branches in the Smalltalk interpreter are represented by actual bytecodes simply as a speed consideration. At the source level of Smalltalk branches are represented as methods selected from a boolean class. However, at the machine level such a representation would be slow, and branching is done sufficiently often to make this a significant consideration.

Variables in the Smalltalk interpreter are implemented as fields within some area of memory; the particular area depends upon the kind of variable. There is a temporary area for method variables, and a global area for class variables. In addition, every object has an area of memory within it for instance variables. The compiler is responsible for ensuring that the correct area field is accessed for any particular variable.

3.2 The Lawine Double Stack Machine

The abstract machine designed to run Lawine programs [Swierstra 1979; Swierstra 1980] is an extension of the standard stack architecture. Typically, a machine has a single stack. On such machines a compiler must use that one stack to hold all the information needed for functions: linkage information for calling and returning from the function, parameters, local variables, and the function result. If the language definition is sufficiently restrictive, as Pascal's is, the size of each piece of function information is of fixed size. However, if the parameters and return value can be of any size then this system is cumbersome. When a function evaluation has been completed, the top of the stack must contain the return value from the function. Space must be reserved for the return value before the call is done so that the linkage information may also be put on the stack for the call, and easily popped off after the function evaluation. Thus, it is necessary to reserve space for the return value at each call of a function, leading to a great deal of duplication of code if a function is called several times.

The problems of a single stack machine can be avoided by using a double stack model. One stack, called the *incarnation* stack, is used to hold the linkage

information and parameters for function calls. The other stack, the *arithmetic* stack, is used to hold arguments to functions, and the result of function calls. Calling a function involves evaluating the arguments on the arithmetic stack, reserving space on the arithmetic stack for the return value, copying the parameters from the arithmetic stack to the incarnation stack, placing the linkage information in the incarnation stack, and branching to the code for the function body.

Problems encountered with such a double stack model are the extra copying of parameters and the necessity of reserving space for function results. The Lawine machine overcomes these problems by having the stacks temporarily switch roles: the incarnation stack becomes the arithmetic stack, and vice versa. This switch occurs just before evaluating the function parameters, and the stacks are switched back after the parameters have been loaded. This way the machine loads the parameters onto the original incarnation stack (since it is made to look like the arithmetic stack), and switches the roles back so that the parameters are actually on the incarnation stack of the function invocation. With this scheme no space is required on the arithmetic stack of the function call to evaluate the arguments and there is no need to reserve space on the arithmetic stack before the call is done. The function result can be put directly on the arithmetic stack without having to worry about over-writing any of the arguments.

3.3 The Maple Machine

The design of the Maple abstract machine was strongly influenced by the machine designs for Smalltalk and Lawine. In Maple, like Smalltalk, the machine consists of of two main portions: a storage manager and an interpreter for Maple opcodes. Since the architecture described in this section is a first attempt at a design for the Maple abstract machine, it includes several deliberate simplifications.

3.3.1 The Maple Machine Storage Manager

An attempt was made to design the storage manager for the Maple Machine along the lines of the Smalltalk Machine, i.e. to base it on some elementary data entity of the Maple programming language. Maple, however, has two basic data structures: groups and elements. It is not clear how to design the Maple storage

manager based on either of these. Consequently, the basic unit of the Maple Machine memory is simply a word, and all Maple groups and elements are composed of *blocks* of words. A block may contain any number of words, and each word may be an instruction, a pointer, or data. A good design for the Maple memory would incorporate the hierarchical nature of the Maple Tree environment. Such a hierarchical storage scheme is left as a future research topic.

The Maple Tree is stored in a so-called 'static' part the memory of the Maple machine. It is called 'static' because the contents of that section of memory may not be changed at runtime. Only the Editor is allowed to manipulate the structure of the Tree, although a running program is allowed to change the value of variable element nodes in the Tree. Every node of the stored Maple Tree is represented by a block of memory, and (except for the root) has a pointer to the father of that node. Every internal node of the Tree has pointers to its field values.

It is the responsibility of the Maple storage manager to allocate and deallocate memory in blocks of words at a time. The Maple Editor and Compiler, when they are designed, will require the storage manager to allocate and deallocate memory permanently. When an expression is evaluated at runtime, the storage manager needs to allocate memory to store temporary values, and then deallocate that memory later when it is no longer required.

3.3.2 The Maple Machine Interpreter

The Maple interpreter is an abstract machine architecture for executing Maple opcodes. The machine design was motivated by the design of the Lawine machine and is an extended stack machine. The Maple machine has one stack, the Result Stack (RS), for storing the results of expression evaluations, and two stacks which grow and shrink by blocks at a time, called Perm and Temp. The values placed on RS are actually pointers to blocks on Perm or Temp which contain the values of expressions. Associated with these three stacks are three registers, RSP, PSP and TSP, which point to the tops of RS, Perm, and Temp, respectively. The machine also has program counter, PC, which points to the current instruction to be executed by the interpreter. Finally, the machine has two registers, ENV and ROOT, which are used to point to particular parts of the Maple Tree. ENV points to the

current environment frame of the running Maple system and is used to find names in the Maple Tree relative to the current semantic environment at runtime. (Environment frames are discussed in detail below.) ROOT points to the top of the Maple tree.

The Maple opcodes emitted by the compiler instruct the interpreter to perform the following general actions:

a. follow a name path through the Maple Tree,
b. allocate a block of memory for a new element or group,
c. branch (conditionally or unconditionally),
d. apply a function to an argument,
e. execute a use clause,
f. execute a case clause,
g. store a value in a variable,
h. push a value onto RS,
i. pop a value from RS,
j. perform simple integer arithmetic,
k. perform garbage collection.

Runtime Garbage Collection

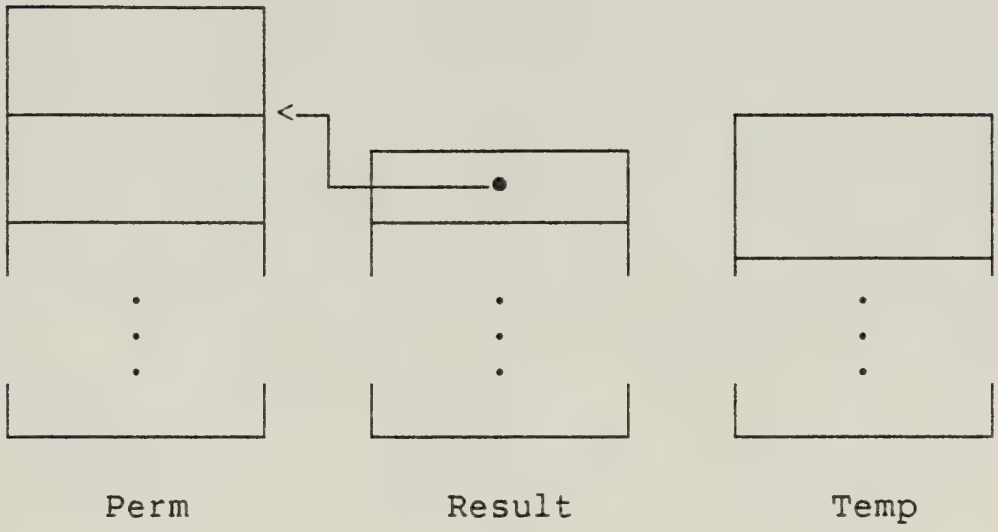
Normally, garbage collection is the responsibility of the storage manager but, due to the design of the Maple Machine architecture, the interpreter handles garbage collection in collaboration with the storage manager. This garbage collection scheme was inspired by the double stack model of the Lawine machine.

The stacks Perm and Temp are used to store the results of evaluating Maple expressions at runtime. Evaluation of expressions on these stacks is defined in such a way that garbage collection can be done easily at runtime. Whenever some expression E is evaluated, the block of memory which contains the value of E is placed on the top of Perm, and the top of RS is made to point to the beginning of this block. Any expression values which are not needed permanently are evaluated on Temp. It is important to note that values in the Maple Machine can be of any size, and so Perm and Temp will grow by varying amounts. The pointers placed on RS are absolute addresses into the stacks Perm and Temp.

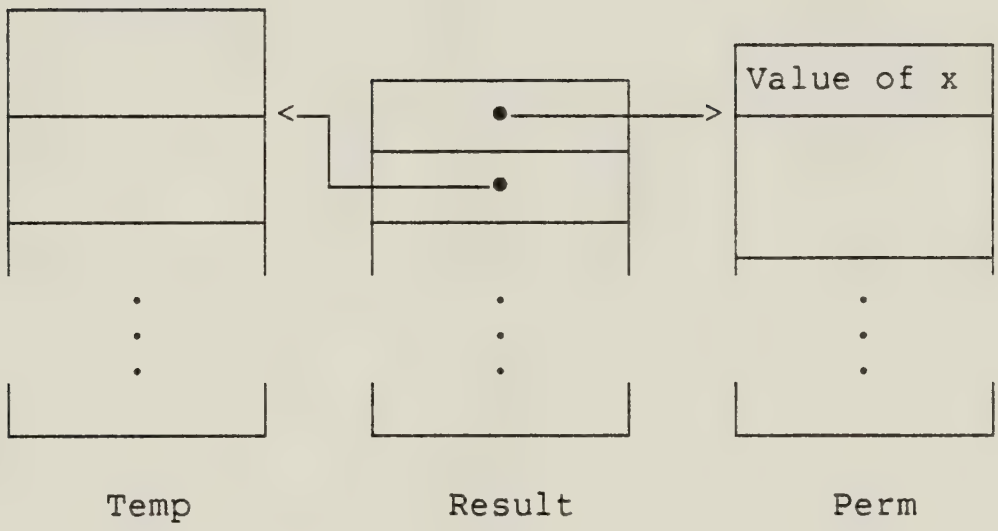
Due to the design of the Maple language, the compiler can, in certain circumstances, determine whether or not some intermediate value, E', which is needed during the evaluation of expression E is required after E has been finished. Such a circumstance is a **use** clause which declares a local counter; the counter is not needed after the evaluation of the clause. In such cases, the interpreter is instructed to **switch** the roles of the stacks so that the Perm stack temporarily behaves as the Temp stack and vice versa.

Consider the nested function applications 'f(g(x))', where function 'f' may discard of its argument after evaluation of the function body, and 'g' may not. The order of evaluation starts with argument 'x' to function 'g', but since the argument to function 'f' may be discarded the stacks Perm and Temp are switched (Figures 3.1(a) and (b)) and TSP is saved in the current environment frame. 'x' is evaluated on the new Perm (since all expressions place their value on the current Perm stack), which is the Temp stack of the evaluation of function 'f', and a pointer to the value of 'x' is pushed on RS (Figure 3.1(b)). Next the evaluation of 'g(x)' is done on Perm (Figure 3.2(a)) and the pointer to this is placed on top of RS (the pointer to 'x' is removed from RS during the function application). Perm and Temp are then reversed back to their original roles for the continuation of the application of function 'f'. The value of 'f(g(x))' is placed on Perm (Figure 3.2(b)) and the top of RS points to that value (again, the pointer to 'g(x)' is removed when 'f' is applied). Finally, by restoring the value of TSP from the environment frame, the temporary values of 'x' and 'g(x)' are popped off Temp (Figure 3.3). Note that if 'g' could have discarded of its argument then the stacks would have been reversed once more before evaluation of 'x', and 'x' would have been placed on the original Perm. Then, after 'g' was complete, the value of 'x' would have been popped off that stack and the evaluation of 'f' would have proceeded as above.

With the above garbage collection scheme there is no need for the storage manager to determine when particular parts of memory are no longer being used (as the Smalltalk storage manager must), provided the compiler can determine which values can be discarded. A full scheme for the compiler to decide what can or cannot be discarded has not been designed, but a simple one has. There are two possibilities when values are created and may be discarded after evaluation of the



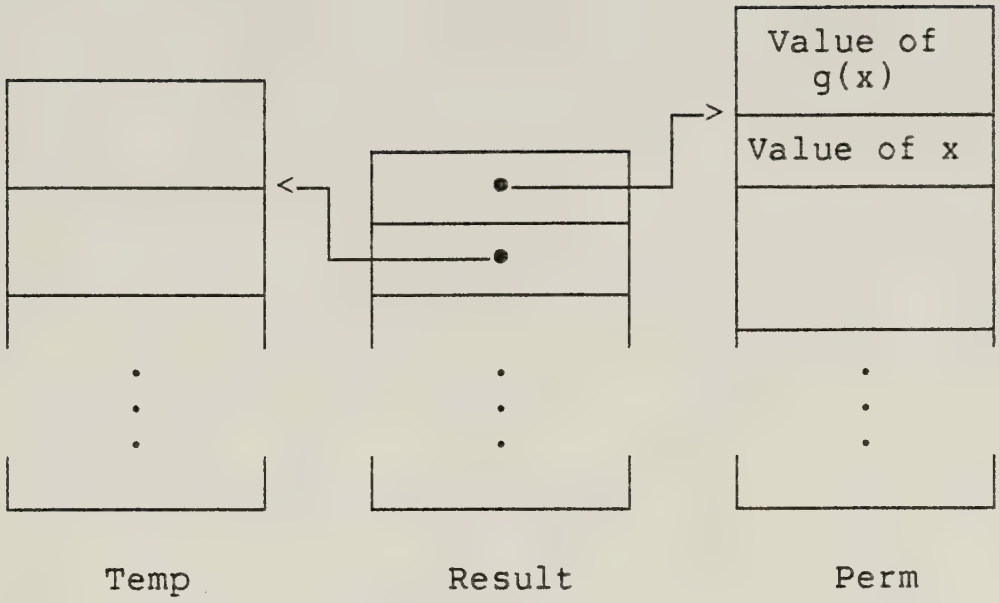
(a)
Before evaluation of f(g(x))



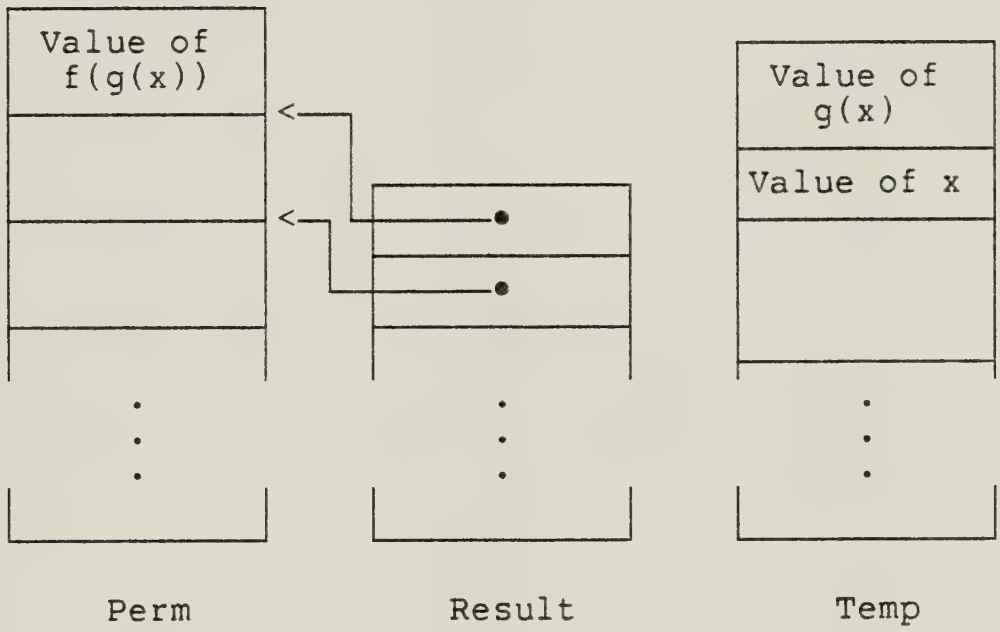
(b)
Stacks switched and x evaluated

Figure 3.1
Runtime Garbage Collection

The evaluation of 'f(g(x))', where f may dispose of its argument and g may not, proceeds (b) by switching the stacks Perm and Temp, and evaluating x on the original Temp (b).



(a)
After evaluation of g(x)



(b)
Stacks switched back and f(g(x)) evaluated

Figure 3.2
Runtime Garbage Collection (continued)

The evaluation of g(x) proceeds on the same stack as x since g may not dispose of its argument (a). The top of RS is replaced by the pointer to g(x). In (b) the stacks have been switched back, and the value of f(g(x)) placed on the original Perm. The pointer to that value has replaced the top of RS.

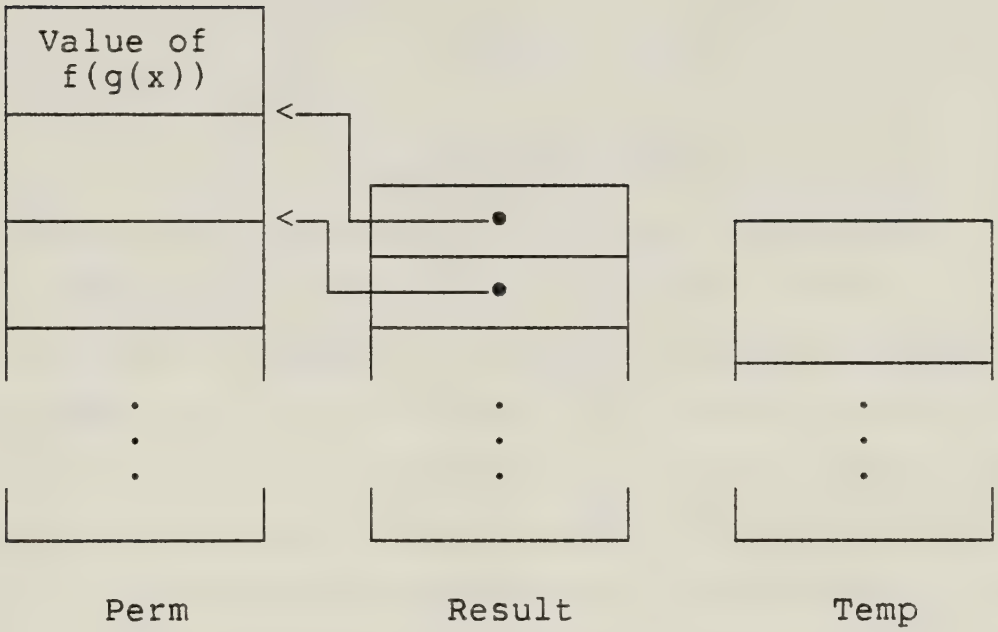


Figure 3.3
Runtime Garbage Collection (continued)

The value of f(g(x)) has been completely evaluated and the temporary values of x and g(x) are popped off Temp.

enclosing construct: a function which returns an element (rather than a group), and a **use** clause which uses an element as its value. In both cases, the argument (or local value) which is created during the evaluation can normally be discarded. A complete scheme for determining which functions and **use** clauses are disposable is left to future research. Some of the difficulties of deciding, automatically, whether something can be discarded are illustrated by the following examples:

```
use g : [ x is int : 5 ; y is char : 'e' ]
in g ;

use h : [ m is var int : int.new 2 ; n is char : 'b' ]
in h ;
```

The first **use** clause declares a group 'g' whose fields are all constants, and returns that constant group as the value of the clause. Since the group which is returned is a constant the local group 'g' does not need to remain after the **use** clause has been evaluated. However, the second **use** declares the group 'h' which contains the variable field 'x'. Since the field 'x' is a variable, the group which is returned by the clause cannot be discarded because it may be changed outside of the **use** clause. That is, the value of the clause may be changed after the evaluation of the clause has been completed because of the variable field. The syntactic, and semantic, differences between these clauses are very slight, but have much influence on the garbage collection. It would appear that it will not be easy for the Compiler to decide exactly what can and cannot be discarded, and that it be necessary to be content with discarding far less than could be discarded.

Environment Frames and Groups

The Maple Tree is stored in the memory of the Maple machine by the Editor and Compiler. The Tree is stored as nodes which are represented by blocks of pointers and values, and the nodes are linked by those pointers to create the hierarchical structure. Every node in the Tree has an UP pointer associated with it which points to its father node. Every internal node of the Tree has a pointer for each of its sons.

When an expression is evaluated at runtime the value is constructed on Perm. The intermediate values created during the evaluation of an expression need to have linkage information associated with them as well since they may be groups too. The Maple interpreter creates an *environment frame* on Perm whenever such

linkage information is required during the evaluation of a Maple expression. An environment frame is a block of memory which contains pointers to other runtime values on Perm (or Temp) or to nodes in the Maple Tree. Each pointer in the block corresponds to a different name which is created during the evaluation. The ENV register of the Maple Machine always points to the first word of the environment frame for the expression which is currently being evaluated. Figure 3.4 shows the general form of an environment frame, although not all environment frames will need or have all the fields shown. The field UP is a pointer to the father node of the expression being evaluated. This only differs from the most recently allocated environment frame when a function application is being executed. For a function application, UP points to the environment of the definition of the function rather than the use of the function.

Field TSP of the environment frame is a word which may be used to store the current pointer to the top of the Temp stack, in case there is a disposable function or **use** clause to be executed in this environment. Field ARG is used only when a function or **use** clause is being evaluated. For a function, ARG points to the block of memory which contains the argument to the function. For a **use** clause, ARG points to the local value declared for the clause. Note that in either case ARG will point to an area on Perm or on Temp, depending on whether or not the associated value may be discarded later. RET is used only for functions; it contains the address to return to after a function has been evaluated. PREV is also used only for functions; it points to the environment frame of the invocation of the function, as opposed to UP, which points to the environment of the definition of the function. PREV is used to restore the runtime environment frame when the evaluation of the function is finished. Finally, ELEM is only used for element functions. For these, ELEM points to the representation of the element from which the function was selected.

A name path is traced up through the Maple Tree, or the runtime intermediate expressions, by following the UP pointers of the Tree nodes, or the environment frames. Following a path down through the Tree is done slightly differently. A Maple Tree node is a block containing pointers to its fields, and an UP pointer. When a new group is created during the evaluation of an expression, the

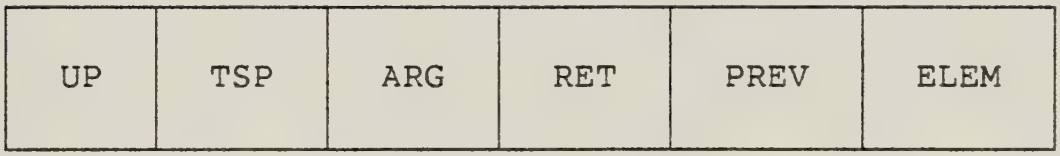


Figure 3.4
General Form of Environment Frames

Each field of an environment frame occupies a single word of storage. Not all environment frames contain all of the fields shown here, althought the existance of any field in a particular frame implies the existance of the fields to its left in this diagram. Fields UP and TSP are always present in an environment frame, while the other fields are optional depending upon what kind of expression is being evaluated.

interpreter is instructed to allocate a new block of memory on Perm. If the group has n fields then the block contains n+1 words. The first word of the block contains the UP pointer to the block representing the group which contains this group. The rest of the words of the block contain pointers to the blocks where the values of the fields of the group are stored. The only exceptions are field values which only require one word to store. In such cases the value is stored directly in the word of the the block representing the group. For instance, the group

```
g : [ a is int : 5 ;
      b : [ c is int: 1; d is int : 2 ]
    ]
```

is stored as a block of three words containing an **up** pointer, an integer 5 value for 'a', and a pointer to another block for 'b'. The block for 'b' would again contain three words: an **up** pointer and integer values 1 and 2 for 'c' and 'd' respectively. The path from 'g' to 'c', 'g.b.c', is followed by selecting the b field of the block for 'g', following that pointer to the block representing 'b' and selecting the 'c' field from that block.

In fact, following the **up** pointers of environment frames, blocks, or Tree nodes, is just a particular example of the general case of following any name path. In general, the the compiler emits code which informs the interpreter to take the i'th field of a block and follow the pointer located in that word. By keeping track of the result of following the last pointer this can continue until the appropriate name location is found. Since the Machine has no knowledge of whether a particular word is an instruction, a pointer, or just data, the compiler is responsible for ensuring that any word treated like a pointer to a block of memory actually is a pointer.

Representation of Classes and Elements

The Maple Machine represents classes and elements as distinct entities. A class can be thought of as a template for the structure of the elements of the class. Since the functions associated with a class cannot be changed at run-time there is no point in allocating storage for those functions for each element of the class. Thus, no element storage representation contains any function bodies. The functions are compiled into a 'static' part of memory (separate from the rest of the Tree), and the fields reserved for functions in blocks are filled with pointers to those

static definitions. (Actually, the same holds true for groups which have function fields.) An element is represented solely by the an value from the base type of the class for that element.

For example, the class:

```
x : [ type is sort [ a with x.type . . . ;
                     b with real . . . ;
                   ]
            as int;
      c with . . . ;
      d with . . .
    ]
```

would be represented by a statically allocated block containing four words (as shown in Figure 3.5): the **up** pointer, a pointer to a block representing the **sort** group, and pointers to the code for 'c' and 'd'. The block representing the **sort** group would contain pointers to the code for functions 'a' and 'b'. An element of this class, which would be allocated dynamically at runtime, would be represented by a block containing an **up** pointer and and integer value from the implicit set for that class. Since an element of this class requires just one word to store the integer value, it would be stored directly in the block containing the element instead in a separate block be itself.

To select a function field from an element the compiler has to do extra work since the function fields are not stored with the element. The compiler emits code to select the function bodies from the class function representation, in the 'static' part of memory, rather than selecting the functions from an element.

Variable elements are represented by pointers to the storage locations they occupy. It is the responsibility of the compiler to ensure that these pointers point to the correct blocks of memory. It is also the responsibility of the compiler to ensure that constant elements do not have their fields changed.

Records

As described in Chapter 2, a record is equivalent to a class definition. The compiler actually builds a class, in the Maple Tree, when a record is declared. For example, the record r : **rec** [x **is** int ; y **is** int] is translated into the following class type:

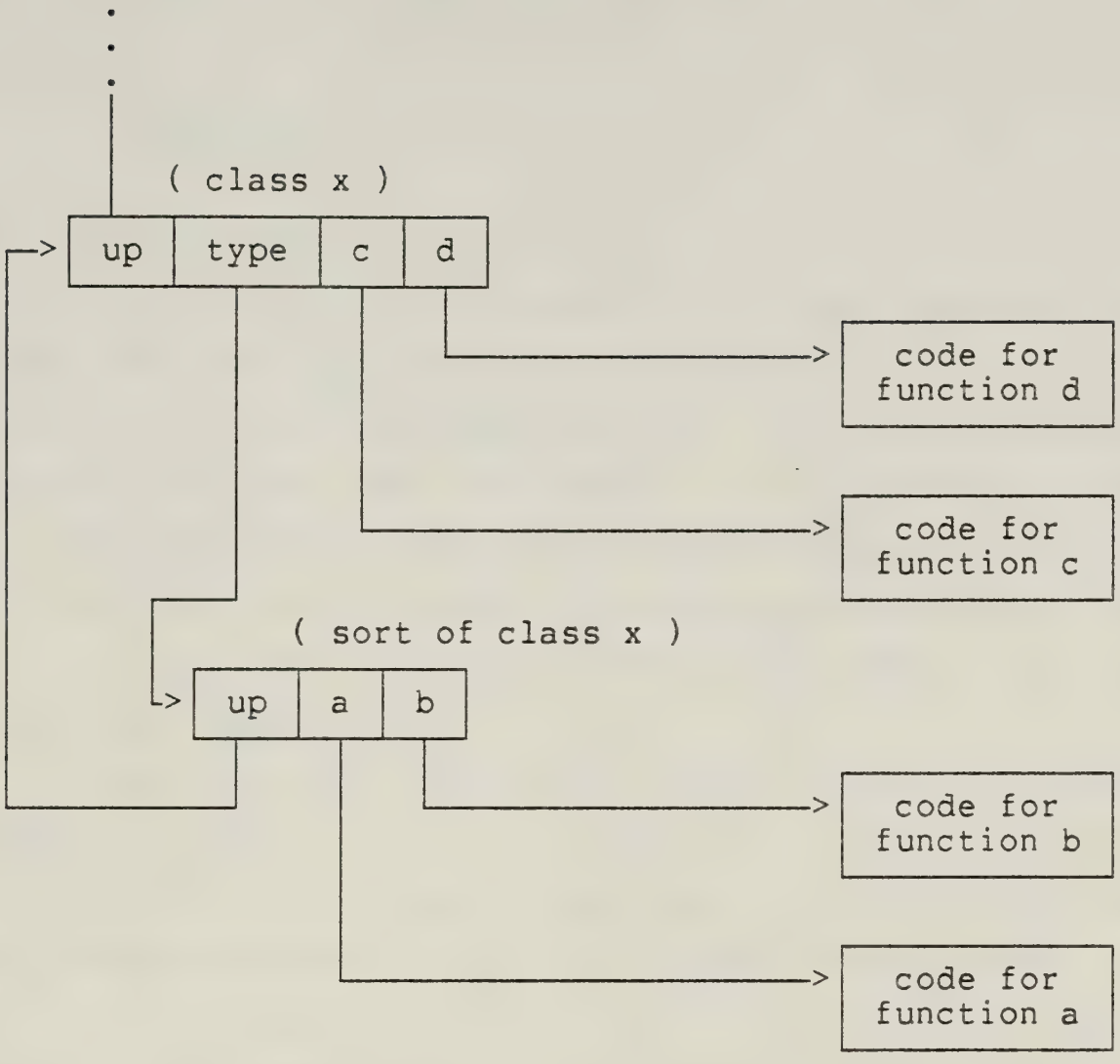


Figure 3.5
The Storage of a Class

The class 'x' given on the preceding page is stored in the memory of the Maple abstract machine as a block containing four pointers. These pointers are to other blocks of memory containing either function code or group representations.

```
r is [ type is sort [ := with r.type is [ ] alter;
                      ? with r.type is order;
                      x is var int;
                      y is var int
                    ];
       new with [ ] is var r.type;
       coerce with [ x, y is int ] is r.type;
     ]
```

and creates the code for the functions :=, ?, new, and coerce. The bodies of these four functions cannot be specified in the Maple programming language because they require explicit knowledge of the underlying machine architecture. The compiler also constructs the implementation type for the class. The implementation type contains the same fields as the fixed part of the group of the **rec** definition. The variant fields are also present in the implementation type, along with a field to hold the current state of the element. The compiler defines the function bodies of :=, ?, new and coerce recursively in terms of the constituent types of the implementation type.

The **rec** construct uses the low level, primitive functions of the machine to manipulate the storage of elements. The functions :=, ?, new, and coerce are defined using the primitive functions of the machine. As a record class is built up these functions are defined in terms of the functions of the constituent types of the record. For instance, the functions for the record r defined above are defined in terms of the functions of 'int', which are supplied by the Maple Tree.

If a record uses variant fields then any elements of that record have enough space allocated to store all the fields of the fixed part, a variant tag, and enough space for variant whose fields occupy the most storage. The variant tags are mapped onto the integers by the compiler and there is one field in the block for the tag field of the record. The value in the tag field of the allocated block indicates the state of the element. It is the responsibility of the compiler to ensure that the proper fields are accessed according to the state of the element. The machine has no knowledge of what fields can be legally accessed at any time, so the compiler has a big responsibility in this area.

The assignment function, ':=', of a record is defined simply by applying the ':=' of each field of the implementation type on each word of the block representing an element of that record. Note that the ':=' function of each of these fields can again be defined in terms of smaller records. For instance, the ':=' of record r

above is defined in terms of the ':=' for int (which is a primitive operation of the machine) on the two fields x and y of the implementation type.

The comparison function, '?', of a record is also defined relatively simply in terms of the constituent fields of the implementation type. Recall that there is a lexicographic ordering defined on all records, in the order of the specification of the fields in the group of the **rec** definition. Thus, '?' is defined as a lexicographic comparison of two records a and b with n fields:

```
if field1 of a < field1 of b then
    a < b
else if field1 of a > field1 of b then
    a > b
else if field2 of a < field2 of b then
    a < b
else if field2 of a > field2 of b then
    a > b

 . . . .

else if field n of a < field n of b then
    a < b
else if fieldn of a > fieldn of b then
    a > b
else a = b
```

The comparison of a field of 'a' with a field of 'b' is defined by the '?' function of the type of those fields.

Function 'new', which creates a new variable element of the record type, allocates a block of memory large enough to store an element of the record class. The variable element returned by new is represented as a pointer to the block of memory allocated by the storage manager.

Finally, the function 'coerce' creates a constant of the record type and assigns a value to it immediately. This is done by allocating a block of memory for an element of the record class, just as new does. The definition of coerce is recursive in the types of the individual fields of the implementation type, and uses the ':=' functions to put the values into the appropriate fields.

3.4 Summary

The design of the Maple abstract machine has been described, and it has been shown how some of the design decisions were influenced by the abstract machine designs of Smalltalk and Lawine. It has laos been shown how the high-level

constructs of the Maple programming language are mapped onto the machine constructs. Finally, the machine representations of the data structures (groups, classes, elements, and functions) have been described. An actual implementation of the Maple abstract machine is described next.

Chapter 4

An Implementation of the Maple Machine

The design outlined in Chapter 3 leaves many details of the Maple Machine to be resolved by the implementation. For instance, the storage manager can be implemented as a virtual memory system. The choice of virtual memory influences everything from runtime memory allocation to stack linkage. This chapter presents a pilot implementation of the Maple Machine. Its storage manager does employ employ virtual memory, leading to the name Maple Virtual Machine for the entire package. The machine implementation was influenced primarily by the Smalltalk Virtual Machine. As was the case with the machine design, the machine implementation includes many simplifications.

4.1 Virtual Memory

The amount of main memory addressable by computers (especially microcomputers) has grown rapidly in recent years, but still doesn't seem to satisfy the needs of programmers. Fortunately, through the use of virtual memory [Denning 1970; Gear 1974], a computer system can be made to appear to have more main memory than the computer is physically capable of addressing.

Virtual memory is a system whereby a program may refer to memory locations which are not actually in main memory. These extra memory locations are on a secondary storage device, typically a disk of some sort. The virtual memory is usually divided into fixed sized *pages* and main memory will contain a certain number of these pages, while the rest of the pages are in secondary store. Any particular virtual memory page can be in main memory at any time and it is the job of the virtual memory system to decide which pages should be in main memory. If a page which is required by a program is currently in secondary store then the virtual memory system must read the page into main memory. Since the number of pages which may reside in main memory is limited, it is necessary to move a main memory page back to secondary storage before the newly requested page may be

brought in; this is referred to as *paging*. The page which is swapped out to secondary store is chosen according to the *paging algorithm* of the particular virtual memory system.

Virtual memory systems are quite efficient because computer programs tend to refer to memory locations which are physically close to each other. Most references within any virtual memory page are to locations also within that page, if the page size is chosen large enough. However, if the page size is chosen too large then pages may contain many unrelated words and main memory space is wasted by the words unrelated to the word(s) responsible for the pages being swapped in. A virtual memory system tries to take advantage of this *locality of reference* by keeping pages which have been mostly recently used in main memory so that the expected future references to those pages can be resolved quickly.

4.2 The Smalltalk-80 Virtual Machine

The Smalltalk-80 Virtual Machine [Krasner 1981; Kaehler 1981] is a virtual memory machine implemented in software on a Xerox microcomputer. The entire Smalltalk Virtual Machine occupies approximately 12K bytes with the virtual memory system accounting for about 40% of that total; the interpreter and the primitive routines make up the rest.

4.2.1 Smalltalk-80 Virtual Memory

Unlike most virtual memory systems which are based upon data which is segmented into fixed sized pages, the Smalltalk Virtual Machine is based upon Smalltalk object descriptions (representations of objects on the machine). The Smalltalk system swaps object descriptions (which are of varying sizes) rather than pages of memory. Since the object descriptions are not of uniform size the Smalltalk system must perform a lot of bookkeeping to keep track of exactly where each object description is located. The object-oriented virtual memory was adopted in order that the Smalltalk Virtual Machine would reflect the structure of the Smalltalk language and take advantage of the strong locality of reference within Smalltalk objects. Since any of the data words contained by an object are highly related there is little waste of space in main memory.

The contents of main memory are maintained by a storage manager called OOZE (Object-Oriented Zoned Environment). Every object in the Smalltalk system is represented by a 16 bit object descriptor allowing for a maximum of 64K unique objects. OOZE maintains a hash table of object pointers which point to the location in main memory of the object they represent. When it was discovered that OOZE spent an inordinate amount of time in hashing, some optimizations were applied. These optimizations included moving the hashing function to microcode, and having the system note the the addresses of frequently used objects.

If the hash table does not contain an entry for a particular object then OOZE must locate that object on disk. The high order bits of an object descriptor are used to locate the object on disk, since all objects with the same high order bits are stored sequentially on disk. The low order bits of the object descriptor indicate the offset of that object within the sequence of objects indicated by the high order bits.

The swapping performed by the Smalltalk system is slightly more complicated than that of a more traditional virtual memory system because not all objects are the same size. As objects are swapped in and out, main memory becomes fragmented. Occasionally, it is necessary for OOZE to go through main memory and perform a compaction by moving all 'active' memory to one end, merging all the unused memory into one large block and updating the hash table.

OOZE also maintains a *reference count* for each object. This reference count indicates how many other objects still point at that particular object. Whenever an object's reference count becomes zero the memory used by that object is freed by the storage manager.

4.2.2 The Smalltalk-80 Interpreter

The Smalltalk interpreter works in cooperation with the storage manager to execute the bytecodes emitted by the Smalltalk compiler. The general actions of the bytecodes are given in Chapter 3.

Various of the high-level operations of the Smalltalk language are implemented directly as primitive operations on the Smalltalk-80 Virtual Machine. The primitive operations of primary importance are integer arithmetic and the subscripting of

variables, which are implemented as bytecodes on the Smalltalk-80 machine. Primitive operations also include the graphic operations to manipulate the bitmap screen quickly. These are implemented as primitive routines.

4.3 The Maple Virtual Machine

The Maple Virtual Machine is a program implemented on the UNIX operating system running on a VAX-11/780. The program consists of two primary modules: a virtual memory storage manager written in the C programming language, and an instruction interpreter written in Pascal. Initially, the entire program was to be written in Pascal, but the Pascal implementation used lacks random-access disk input/output. The disk input/output was done in C, but since the interface between Pascal and C is hard to use the whole virtual memory system was implemented in C. The entire program occupies approximately 40K bytes with the interpreter accounting for 80% and the virtual memory storage manager accounting for the rest. The virtual memory module of the Maple machine is small, relative to the interpreter, because the C compiler emits more efficient code on the VAX than the Pascal compiler.

The reason for the vast difference in size between the Maple Virtual Machine program and the Smalltalk-80 program is that the VAX is a 32-bit machine while the Xerox microcomputer is a 16-bit machine. Thus, an instruction on the Xerox machine only occupies one half the space of a VAX instruction. The more powerful instruction set of the VAX only partially compensates for the space imbalance.

4.3.1 The Maple Virtual Memory

The Maple storage manager was designed as a standard virtual memory system based on pages of words of memory. The entire Maple Tree, including all functions and data, is represented in virtual memory. One section of virtual memory is reserved as 'static' memory which does not change at runtime. When a Maple program is compiled the compiler interacts directly with the storage manager to store all executable code in this static area. Note however, that no assumptions are made about which pages make up the static area, or even whether those pages are

contiguous.

The Maple Virtual Machine has a maximum of 128 pages of virtual memory, each of which contains 256 words. The entire virtual memory is stored as a UNIX file, and when using 128 pages this took over 132K bytes to store. Since none of the test programs run with this program ever approached this small limit on pages, 128 pages were considered sufficient for a first implementation.

Each word of virtual memory is represented as a 32-bit VAX integer. Any particular word may be used to store 32 bits of data, an address (as two 16-bit parts), or an instruction (as two 16-bit parts).

An address in the Maple Virtual Machine consists of a page number and an offset within that page. Since Pascal does not permit access to the bit level, it was not possible to optimize the allocation of bits to these two parts. Consequently, each part takes up a full 16 bits, although the page offset actually requires just 8 bits to store. Not only are there 8 bits wasted, but the number of pages is limited to 65535 ($2^{16} - 1$). This page limit was not a problem in this implementation.

An instruction on the Maple Virtual Machine also consists of two parts: the opcode, and data for the opcode. Note that not all opcodes use the data field of the instruction word. The opcode and data fields are each allocated 16 bits for the same reasons given above for addresses. The opcodes for this implementation fall into the following general categories:

a. memory allocation and deallocation,
b. finding a node in the Maple Tree (name path following),
c. comparisons and branches,
d. function application,
e. evaluation of Maple clauses,
f. integer arithmetic,
g. Result Stack operations.

The opcodes are discussed in more detail later in this chapter. Note that input/output is not included in the above list. An appropriate input/output facility is left for future research.

Each of the interpreter stacks RS, Perm, and Temp, described in Chapter 3, are stored as contiguous virtual memory locations. Rather than reserve some fixed area of memory for these stacks, they are allowed to occupy unlimited pages of virtual memory, which are linked together by pointers. A page which is used to store part of a stack is not allowed to be used for any other kind of data until it is no longer required by that stack. A pointer *next*, at the top of each stack page, gives the page number of the page which precedes it in the stack. The interpreter contains registers which point to the top of each stack. Each page of virtual memory also contains a *side* pointer which is reserved for future use by the Editor and Compiler for making changes to definitions in the Maple Tree.

The virtual memory storage manager allocates memory in blocks of contiguous words and no block is allowed to split over a page boundary. This means that the virtual memory page size must be chosen so that no block will exceed that size. Since blocks are only used as environment frames, groups, or elements, it seems unlikely that the 256 word page used by this implementation will ever be exceeded. Environment frames never exceed 6 words, and it is hard to imagine a programmer creating a group with many more than 20 fields. Since an element is represented as a copy of its implementation type, it seems unlikely that elements could be too large either. If a programmer does create a data type which is too large, it is the responsibility of the compiler to either report an error, or to coerce the type (with the use of pointers) so that it will fit in a page.

Each virtual memory page contains a count of how many of the words within it have been allocated. Since all allocation within a page is done consecutively, starting at the first word of the page, this count also indicates which is the next word in the page which may be allocated.

The storage manager can maintain 8 pages of virtual memory in the main memory of the Maple Virtual Machine. The storage manager keeps a table in main memory which indicates the location of each page of virtual memory and the number of words allocated within each page. This table is not a hash table, as in Smalltalk-80, but is directly indexed by the page number for each page. Hashing was not necessary in this implementation because there are only 128 pages in the entire system, and a table of 128 entries could easily be stored in main memory.

Each of the pages in main memory maintains information for swapping about 1) where it came from on disk, and 2) how the contents of the page have been used while the page has been in main memory. If any word within a page has been changed then the whole page is considered to have been changed. If none of the words within a page have been changed then that page need not be written back to disk when it is swapped out of main memory. This small optimization saves a great deal of unnecessary disk access.

The Maple Virtual Machine uses a modification of the least-recently-used paging algorithm known as the Clock algorithm [Carr and Hennesy 1981]. Whenever a page is swapped into main memory all pages in main memory are marked UNUSED. Then, whenever the contents of a page are referenced, the page is marked USED. When a new page needs to be swapped in, the memory manager looks at each page in main memory for the first one which is still marked UNUSED from the previous swap. If none of the pages are still marked UNUSED then the first page examined is swapped out. The search for an UNUSED page does not always start with the same page, because that page would tend to be swapped very frequently. Instead, the search always starts at the next page after the last page to be swapped, and continues around the list of pages in a cyclic fashion. So, if page 3 has just been swapped then the next time swapping is required, page 4 will be the first candidate considered.

4.3.2 The Maple Virtual Machine Interpreter

The interpreter cooperates with the storage manager to fetch instructions from the static section of virtual memory and execute them. Since the interpreter and storage manager cannot tell whether a particular word represents an instruction, address, or data, the compiler is responsible for ensuring that the code is executed correctly.

As mentioned in Chapter 3 the interpreter maintains six registers. These are represented in the Maple Virtual Machine by 32-bit registers which point at various parts of virtual memory:

a. a program counter, PC,

b. a pointer to the current environment frame, ENV,

c. a pointer to the top of the Result Stack, RSP,

d. a pointer to the top of the Perm Stack, PSP,

e. a pointer to the top of the Temp Stack, TSP,

f. a pointer to the root of the Maple Tree, ROOT.

Since the two stacks Perm and Temp are represented in the interpreter by pointers to their respective tops, the contents of the two stacks may be switched easily by swapping the two registers PSP and TSP.

4.4 Maple Opcodes

The Maple Virtual Machine has over 30 opcodes. Many of these opcodes are for relatively common operations such as branching (6 opcodes) and integer arithmetic (7 opcodes). There are also several opcodes for operations tailored to the Maple langauge such as memory allocation, name path following, function application, **use** clauses, **case** clauses, and garbage collection. A complete list of the opcodes and their execution on the Maple Virtual Machine are given in Appendix B.

4.4.1 Memory Allocation

The opcodes CONS, FIELD, and ENDCONS are used to allocate a new block of memory on top of Perm and to initialize the fields of that block to particular values. CONS allocates a block of memory, the size of which is indicated by the data field of the instruction, stores the ENV register in the first word of the block (as the UP pointer), and sets ENV to point to this block. The opcode FIELD is used to assign a value to a particular field of the block allocated by the most recent CONS. The data field of FIELD is used to indicate which word of the allocated block is to receive the value which is currently on top of RS. The opcode ENDCONS indicates that the fields of the block have been completely initialized, and causes the interpreter to place the pointer to this block on top of RS to represent the 'value' of the current expression. ENDCONS also restores the previous environment pointer back to the ENV register.

The CONS, FIELD's, ENDCONS sequence is used to allocate memory for groups and elements which are created at runtime. Between any CONS–ENDCONS pair

there can be any number of FIELD opcodes, with one expression (encoded in Maple opcodes) for each. There should be as many expressions and associated FIELD instructions as there are fields in the group or element which is being allocated. The final value of a particular word of a block is the last value assigned to it by a FIELD instruction. There is no restriction on the size or complexity of the code for the field expressions.

4.4.2 Name Paths

Nodes in the Maple Tree are found at runtime by following name paths through the Tree. A name path can start relative to either the current position (environment) in the Tree or the top (root) of the tree. The interpreter has two opcodes for these situations: CURR and ROOT. CURR pushes the ENV register on top of RS, while opcode ROOT pushes the ROOT register on RS. In either case, the top of RS then points at a block of memory which contains pointers to other sections of the Maple Tree. The opcode SEL takes the address, A, on the top of RS, pops A off RS, selects the i'th field of the block pointed to by A (where i is the value in the data field of the SEL instruction), and pushes the contents of that field on RS. These SEL instructions are repeated as much as necessary to arrive at the desired node in the Tree.

If the node being found is a function, then an extra instruction is necessary at the end of the sequence of SEL instructions. Enough SEL's are done to find the block in which the function was declared. Recall that functions are not stored with the groups or elements which contain them, but rather they are in a 'static' portion of virtual memory which is not changed at runtime. During the evaluation of a function, **up** is supposed to refer to the environment of the definition of the function. However, at runtime the environment of the usage of the function is not the same as the environment of the definition of the function. Since the pointer to the code of the function is actually stored in the block of the definition of the function, the address of the block containing that pointer is also the **up** pointer for the body of the function. Therefore, once the block containing the function has been found the opcode FSEL is executed. FSEL pushes a copy of the top of RS onto RS (as the **up** pointer for the function body), and replaces the top of RS

with pointer to the code for the function body.

If, on the other hand, a Maple variable which only occupies a single word is being found, a different instruction is used at the end. Recall that such variables are stored directly in the block of their definition. Since variables are represented as pointers to the area of memory containing their value it is necessary to put the address of the variable field on RS. Once the block containing the variable has been found the top of RS contains the address of the first word of the block. All that needs to be done is to add the offset of the variable field to the top of RS and then the top of RS will contain the address of the variable field. This is achieved by the opcode OFFSET which adds its data field to the top of RS.

4.4.3 Function Application

Since there are two kinds of functions, single argument functions and element functions, the interpreter has two kinds of function application instructions. A single argument function application is executed by the following steps. First the address of the code for the function and the environment pointer of the function definition are placed on top of RS (using the name path scheme described above in 4.4.2). Then the argument is evaluated and a pointer to the argument is placed on the top of RS. The opcode APPLY then instructs the interpreter to create a new environment frame to contain the argument pointer, the return address, the environment pointer for the function, and the current ENV. Finally, all the function information on RS is popped off and placed in the environment frame, and the interpreter branches to the code for the function. The compiler is responsible for ensuring that there is a RETURN opcode as the last instruction of the function body. RETURN has the effect of restoring the PC and ENV registers from the environment frame of the function evaluation.

If an element function is being applied then there are some slight changes. First the pointer to the element from which the function is selected is the first thing pushed on RS. Then the function body is found and the argument evaluated as above. The opcode EAPPLY instructs the interpreter to construct an environment frame as before, except it will also contain the element pointer. The evaluation of the function proceeds as before.

In this implementation the function body refers to the argument, through the environment frame, with the code 'CURR ; SEL 3', and the element by 'CURR ; SEL 5'. A function performs its evaluation on Perm and leaves a pointer to the block representing the function value on top of RS. If a function can dispose of its argument after evaluation then the following is done. Before the argument is evaluated, the opcode MARK places a copy of TSP in the current environment frame. Then the opcode SWITCH causes the two stacks Perm and Temp to reverse roles, so that the argument is evaluated on the Temp stack of the function. Once the argument is evaluated another SWITCH reverses the stacks back again. After the APPLY or EAPPLY instruction a DISCARD opcode will cause the argument to be discarded from Temp by restoring the TSP from the environment frame and requesting the storage manager to deallocate the memory used for Temp in the meantime.

4.4.4 Use Clauses

A **use**-clause is implemented similar to a function. The code to evaluate the local value of the **use**-clause is emitted, and the address of the local value is placed on top of RS. The opcode USE causes a new environment frame to be allocated with the address of the local value placed in the ARG field of the environment frame. The code for the body of the use comes next, with the local value refered to by 'CURR ; SEL 3'. Finally, the opcode ENDUSE pushes the pointer to the value of the use-clause on RS and restores the previous ENV register from the environment frame.

A disposable **use**-clause is implemented by a MARK and SWITCH before the code for the local value which causes the local value to be placed on Temp. After the code for the local value, and before the USE instruction, another SWITCH reverses the stacks back again. After the ENDUSE instruction a DISCARD causes the local value to be discarded from Temp.

4.4.5 Case Clauses

A **case**-clause requires more work on the part of the compiler than any of the previous Maple instructions. The compiler compiles the statement lists into

static memory and constructs a block which contains the address of each statement list. The code for the **case**-clause starts with the code to evaluate the discriminator of the clause. Next comes the opcode CASE followed by a word containing the address of the block of statement list addresses. CASE causes the interpreter to use the value on top of RS (the value of the discriminator) as an index into the block of addresses. The interpreter then branches to the first instruction of the appropriate statement list. It is the responsibility of the the compiler to place a branch at the end of each statement list which returns execution to the instruction after the CASE instruction. As well, the compiler must ensure that there are enough entries in the statement list block for every possible value of the discriminator, including the **out** clause.

Since the value of the **case**-clause is the value of the last expression in the expression list which is executed, the intermediate expressions must not leave their values on RS. The opcode POP is used to dispose of the value on top of RS, and the intermediate expressions must each be followed by a POP instruction to remove their values from RS.

4.4.6 Integer Operations

Since the basic unit storage on the Maple Virtual Machine is the 32-bit word there are several opcodes for manipulating words. These opcodes perform integer arithmetic, assignment, and comparisons. All the integer opcodes, except for NEG and ABS, assume that they are invoked as element functions, i.e. the current environment frame contains an argument field, and an element field. The argument field is assumed to contain an integer value rather than a pointer. The element field is assumed to be an integer variable, i.e. a pointer to a word containing an integer value. Since these opcodes are invoked as functions, they all restore the previous ENV register as they return. The opcodes NEG and ABS are single argument functions and, as such, do not have element fields. Note that none of these integer instructions allocate any memory on Perm.

The following integer arithmetic opcodes put their integer value result directly on top of RS:

a. ADD, to add two integer values,

b. SUB, to subtract one number from another,

c. MUL, to multiply two integer values,

d. DIV, to divide two integers and return the quotient,

e. MOD, to return the remainder of dividing one integer by another,

f. NEG, to return the negation of an integer,

g. ABS, to return the absolute value of an integer.

The first integer to these instructions is the element from which the function was chosen. The second integer (or only integer for NEG and ABS) is the argument to the function.

The opcode ASSIGN takes the integer value in the argument and stores it in the element integer variable. ASSIGN also pushes a null value, 0, on RS to represent the null group returned by the assignment function, :=, of the Maple programming language. The opcode DATA is used to push the next word in the code stream on top of RS.

The opcode CMP compares the value of the element variable to the value of the argument, and pushes the result of the comparison on RS. There are three integer constants defined within the interpreter: LessThan, EqualTo, and GreaterThan. The interpreter pushes the appropriate one of these constants on RS depending upon how the value of the element compares to the value of the argument.

The branch opcodes of the Maple Virtual Machine use the top of RS as the condition code of the most recent comparison. If the branch is taken the top of RS is popped, otherwise it is left alone so that further branches may use that condition. The various branch opcodes are:

a. BRA, branch unconditionally,

b. BLT, branch on less than,

c. BLE, branch on less than or equal to,

d. BEQ, branch on equal to,

e. BGE, branch on greater than or equal to,

f. BGT, branch on greater than.

Each of these branch instructions uses the following word in the code stream as the address to branch to.

All records in Maple are built, ultimately, out of these integer opcodes. For instance, the := function of a record is defined recursively in terms of the := functions of the constituent types of the record. But at the lowest level these types must be mapped onto the integers, and they use the ASSIGN opcode to perform the assignment.

4.5 Summary

The details of of an implementation of the Maple abstract machine, called the Maple Virtual Machine, have been presented. Since the Smalltalk-80 Virtual Machine had a strong influence on the implementation some of the details of the Smalltalk machine have been presented. The details of the instruction set for the Maple Virtual Machine have been presented since the Compiler must know the exact workings of the machine.

Chapter 5

Summary

This thesis has presented some of the details of the integrated Maple Programming System, and described the machine which and forms the foundation for the rest of the Maple project. The portion of the machine of primary importance is the runtime garbage collection scheme.

The Maple Programming System is based upon the Maple programming language which is is strongly typed and contains the mechanisms for the definition of some very generalized data types, including groups and classes. The data structures of the Maple language may contain both data and functions as fields. Due to the hierarchical nature of the data structures the entire collection of all Maple programs and data in a Maple Programming System is called the Maple Tree.

In order to run Maple programs efficiently a machine architecture which is tailored to the Maple language is required. The Maple abstract machine, with its double stack scheme for runtime garbage collection, has been described. As well, the representation of the Maple programming language data structures on this abstract machine have been presented. However, many of the details of the machine were not given since they were left for particular implementations to resolve.

Since the design of the abstract machine was significantly different from what has come before, an implementation of the Maple abstract machine was created. The details of this implementation, called the Maple Virtual Machine, have been presented. This machine, implemented on a VAX computer, has successfully run some simple hand-compiled test programs. These test programs were quite slow, which is not surprising considering the whole machine was implemented in software.

5.1 Future Research

The Maple project is fertile with future research topics, foremost of which is the structure of the Maple compiler. As mentioned in Chapter 2, the extended Maple language is highly context-sensitive. The compiler for the extended language will require slightly more complicated parsing techniques than more simple languages such as Pascal. Also, the compiler is responsible for determining which runtime values may be discarded during garbage collection. The full scheme for making this decision has not yet been designed. Other areas which require more work are the exact specification of the structure of the Maple programming system, especially the input/output mechanisms, and the interaction between the various components of the Maple Programming System (Editor, Compiler, Machine). The exact specification of the interactions are necessary because the Machine may need to be slightly altered to conform to the decisions made. For example, the Editor may or may not communicate directly with the storage manager within the Machine. If the Editor does not communicate directly then it must use the Compiler as an intermediary.

The appearance of the Maple Editor to the user is quite important since that is the major interface to the user. A machine-user interface similar to that of Smalltalk [Tesler 1981] would be desirable. Smalltalk has a high-resolution screen, a keyboard, and a 'mouse' for the user to interface with the Smalltalk programming system. The result is that the Smalltalk system is highly user-friendly. When a a system is easy for a user to become acquainted with and use, there is a great deal of encouragement for the user to switch to that system. Thus, it would be best for its acceptance if the Maple Programming System were as user-friendly as Smalltalk's, and the Editor were as easy to interface with.

Since the abstract machine described in this thesis was a pilot implementation, there were many deliberate simplifications made. There is much work which needs to be done to optimize the bit allocation in instructions and addresses in the memory of the machine. Indeed, the instruction set presented here is not necessarily the best possible for running Maple programs; it is not known at present what instructions would be best. As well, the basic design of the virtual memory storage manager might be improved if based upon some unit of storage of the Maple language, especially some hierarchical model.

Finally, the entire Maple project must be continually updated to handle further refinements made to the Maple programming language.

References

Barnes, J. P. G., 1980. An Overview of Ada. *Software - Practice and Experience*, Volume 10, 1980: 851 - 888.

Boehm, H., Demers, A., Donahue, J. 1980. *An Informal Description of Russell.* Technical Report 80-430, Department of Computer Science, Cornell University, Ithaca, New York.

Carr, R.W., Hennessy, J.L., 1981. *Proceedings of the Eighth Symposium on Operating Systems Principles*, Pacific Grove, California, December 1981: 87-95.

Dahl, O., Birtwistle, G., Myhrhaug, B., Nygaard, K., 1973. *Simula Begin.* Philadelphia: Auerbach, 1973.

Demers, A., Donahue, J. 1979. *Revised Report on Russell.* Technical Report 79-389, Department of Computer Science, Cornell University, Ithaca, New York.

Denning, P. J. 1970. Virtual Memory, *Computing Surveys* Volume 2, Number 3: 153-189.

Gear, W.C. 1974. *Computer Organization and Programming, 2nd Edition.* McGraw-Hill Book Company, New York, New York.

Goldberg, A. 1981. Introducing the Smalltalk-80 System. *BYTE* Volume 6, Number 8: 14-26.

Ingalls, D.H.H. 1978. The Smalltalk-76 Programming System: Design and Implementation. *Proceedings of the Fifth Annual Conference on Principles of Programming Languages*, January 1978: 9 - 16.

Jensen, K., Wirth, N., 1975. *Pascal User Manual and Report, 2nd Edition*, Springer-Verlag, New York, New York.

Kaehler, T. 1981. Virtual Memory for an Object-Oriented Language. *BYTE* Volume 6, Number 8: 378-387.

Krasner, G. 1981. The Smalltalk-80 Virtual Machine. *BYTE* Volume 6, Number 8: 300-320.

McCarthy, J., 1960. Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I. *Communications of the ACM* Volume 3, Number 2: 184–195.

Pagan, F.G., 1976. *A Practical Guide to Algol68*, Wiley and Sons, Toronto, Ontario.

Ritchie, D.M., Thomson, K., 1974. The Unix Time-Sharing System. *Communications of the ACM.* Volume 17, Number 7, 1974: 365–375.

Swierstra, S.D. 1979. *Machine Architectures for Block-Structured Languages.* Memorandum 262, Department of Applied Mathematics, Twente University of Technology, Netherlands.

Swierstra, S.D. 1980. *Lawine, An Experiment in Language and Machine Design.* Ph.D. Dissertation, Department of Applied Mathematics, Twente University of Technology, Netherlands.

Tennent, R.D. 1981. *Principles of Programming Languages.* Prentice/Hall International, Incorporated, London, England.

Tesler, L. 1981. The Smalltalk Environment. *BYTE* Volume 6, Number 8: 90–147.

Voda, P.J. 1982a. Maple: A Programming Language and Operating System. *Proceedings of the Ninth Annual ACM Symposium on Principles of Programming Languages*, January 1982: 157–168.

Voda, P.J. 1982b. *Maple: A Total Modular Environment, the Maple Language.* Technical Report (in production), Department of Computing Science, University of Alberta.

The Xerox Learning Research Group 1981. The Smalltalk-80 System. *BYTE* Volume 6, Number 8: 36–47.

Appendix A

A Grammar for the Maple Programming Language

This appendix gives a grammar for the strict Maple programming language. The extended Maple language is formed from this grammar by applying the abbreviations described in [Voda 1982b]. The syntax production names should be self-explanatory. Brace brackets are for meta-grouping repetitions, alternatives or optional phrases. The symbol "*" following such a grouping indicates zero-or-more-times repetition, and "+" indicates one-or-more-times repetition. The symbol "|" is used to separate alternative phrases within a grouping. Finally, when a set of brace brackets do not contain a repetition or set of alternatives they enclose an optional phrase. All non-terminal production names start with a capital letter, and all terminal symbols are in boldface type. When the symbols "|","*", and "+" are part of a production they are enclosed in double quotes.

The Maple Tree is the basic structure of a Maple system. The Maple Tree itself is a group containing other groups, classes, functions, and elements. However, in general a Maple program is an expression:

Expression = Group | Selection | Clause | (Expression)

Group = [< Field ; }*] | Record

```
Field = Selector { ProperField | FunctionField |
                   SortField }
ProperField = is FieldType { : FieldValue }
FieldType = Type
FieldValue = Expression
Type = Group | ElementType
FunctionField = with ArgumentType ProperField
ArgumentType = Type
SortField = is SortType : SortValue
SortType = Sort
SortValue = Sort
Selection = GroupSelection | ElementSelection
GroupSelection = Subject . Selector { Argument }
Subject = env | GroupExpression
Selector = StandardSelector | Word | Operator |
           CharacterSelector
StandardSelector = arg | elem | tag | up
Word = Letter { Letter | Digit | _ }*
Letter = a | . . . | z | A | . . . | Z
Digit = 0 | . . . | 9
Operator = { SimpleOperator }+ | :=
CharacterSelector = ' { Letter | Digit | MapleSymbol |
                       SimpleOperator | Space | _ } '
MapleSymbol = ' | " | : | [ | ] | ( | ) | . | , | ; | "|"
GroupExpression = Expression
SortExpression = ClassSelection . Selector
ClassSelection = Subject
```

```
Record = rec RecordGroup
RecordGroup = ApparentGroup
Clause = CaseClause | AttemptClause | ParallelClause |
         FailClause
CaseClause = case Discriminator in CaseTail
Discriminator = Expression
AttemptClause = attempt ExpressionList else CaseTail
CaseTail = { Alternative }* OutPart
Alternative = "|" Selector ExpressionList
OutPart = out ExpressionList end
ExpressionList = Expression ; { Expression ; }*
ParallelClause = par Process ; { Process ; }* end
Process = Expression
FailClause = fail < Exception }
Exception = Selector
Sort = { shared } sort ApparentGroup { Implementation }
ApparentGroup = [ { FixedPart }* { Variants }* ]
FixedPart = ApparentField ;
ApparentField = Selector { with ArgumentType } is
                ApparentFieldType { SideEffect }
                { : ApparentFieldValue }
ApparentFieldType = [ ] | ElementType
ApparentFieldValue = Expression
SideEffect = { VariantTag }* alter
VariantTag = "|" Selector
Variants = VariantTag FixedPart
Implementation = as SortSelection
```

```
ElementType = { VariantTag }* { var } SortSelection

ElementSelection = Element . Selector { Argument }

Element = Expression
```

Appendix B

Maple Virtual Machine Opcodes

Every Maple Virtual Machine opcode is made up of one or more meta-operations on the actual machine. This appendix describes how the opcodes given in Chapter 4 are implemented on the interpreter. The format of each opcode is given, along with a description of how the interpreter executes it. The various Maple Virtual machine meta-operations used in the descriptions are:

a. '←' is used as the assignment operator,

b. *top* refers to the top of RS,

c. *push* pushes the next word on RS,

d. *pop* pops the top of RS off,

e. square brackets are used to show which opcodes use their data field,

f. 'alloc m' means allocate a block of m words on the top of Perm and return the address of that block,

g. t represents some word of temporary storage within the machine,

h. 'rel x y' means deallocate all memory between address x and address y,

i. X(i) is indexed addressing. The contents of X are used as a pointer to a block of memory, and the i'th

word is selected from that block. X(0) is the word which the contents of X point to.

The description of the opcodes starts with memory allocation:

```
[ CONS n ] :==

    t ← alloc (n+1)
    t(0) ← ENV          *** up pointer
    ENV ← t


[ FIELD i ] :==

    ENV(i) ← top
    pop


ENDCONS :==

    push ENV
    ENV ← ENV(0)


CURR :==

    push ENV


ROOT :==

    push ROOT


[ SEL i ] :==

    top ← top(i)


[ FSEL i ] :==

    push top
    top ← top(i)


[ OFFSET i ] :==

    top ← top + i
```

```
APPLY :==

   t ← alloc 5          *** new environment frame
   t(4) ← ENV           *** save previous environment
   t(2) ← top           *** save argument pointer
   pop
   t(3) ← PC            *** save return address
   PC ← top             *** prepare to branch to function
                        code
   pop
   t(0) ← top           *** save up pointer for function
                        body
   pop
   ENV ← t


EAPPLY :==

   t ← alloc 6          *** new environment frame
   t(4) ← ENV           *** save previous environment
   t(2) ← top           *** save argument pointer
   pop
   t(3) ← PC            *** save return address
   PC ← top             *** prepare to branch to function
                        code
   pop
   t(0) ← top           *** save up pointer for function
                        body
   pop
   t(5) ← top           *** save the pointer to the element
   pop
   ENV ← t


RETURN :==

   PC ← ENV(3)          *** restore return address
   ENV ← ENV(4)


MARK :==

   ENV(1) ← TSP


SWITCH :==

   t ← PSP
   PSP ← TSP
   TSP ← t
```

```
DISCARD :==

   rel ENV(1) TSP
   TSP ← ENV(1)


USE :==

   t ← alloc 3
   t(0) ← ENV
   t(2) ← top          *** save pointer to local value
   pop
   ENV ← t


ENDUSE :==

   ENV ← ENV(0)


CASE addr :==

   PC ← addr(top)
   pop


POP :==

   pop


ADD :==

   t ← ENV(5)          *** address of element
   t ← t(0) + ENV(2)
   push t


SUB :==

   t ← ENV(5)          *** address of element
   t ← t(0) - ENV(2)
   push t


MUL :==

   t ← ENV(5)          *** address of element
   t ← t(0) * ENV(2)
   push t
```

```
DIV :==

    t ← ENV(5)          *** address of element
    t ← t(0) div ENV(2)
    push t


MOD :==

    t ← ENV(5)          *** address of element
    t ← t(0) mod ENV(2)
    push t


NEG :==

    t ← - ENV(2)
    push t


ABS :==

    t ← abs ENV(2)
    push t


ADD :==

    t ← ENV(5)          *** address of element
    t ← t(0) + ENV(2)
    push t


CMP :==

    t ← ENV(5)
    if t(0) < ENV(2) then push LessThan
    elif t(0) = ENV(2) then push EqualTo
    else push GreaterThan
    fi


BRA addr :==

    PC ← addr
    pop


BLT addr :==

    if top = LessThan then PC ← addr; pop fi
```

```
BLE addr :==

   if top = LessThan or top = EqualTo then PC ← addr; pop
   fi


BEQ addr :==

   if top = EqualTo then PC ← addr; pop fi


BGE addr :==

   if top = GreaterThan or top = EqualTo then PC ← addr;
   pop fi


BGT addr :==

   if top = GreaterThan then PC ← addr; pop fi


ASSIGN :==

   t ← ENV(5)
   ENV(0) ← ENV(2)


DATA value :==

   push value
```